

数据科学家成长之路

Python大战机器学习

数据科学家的第一个小目标

华校专 王正林 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

VISIT…

LANZAROTE Caliente.com

内 容 简 介

数据科学家是当下炙手可热的职业，机器学习则是他们的必备技能。机器学习在大数据分析中居于核心地位，在互联网、金融保险、制造业、零售业、医疗等产业领域发挥了越来越大的作用且日益受到关注。

Python 是最好最热门的编程语言之一，以简单易学、应用广泛、类库强大而著称，是实现机器学习算法的首选语言。

本书以快速上手、四分理论六分实践为出发点，讲述机器学习的算法和 Python 编程实践，采用“原理笔记精华 + 算法 Python 实现 + 问题实例 + 代码实战 + 运行调参”的形式展开，理论与实践结合，算法原理与编程实战并重。

全书从内容上分为 13 章分 4 篇展开：第一篇：机器学习基础篇（第 1~6 章），讲述机器学习的基础算法，包括线性模型、决策树、贝叶斯分类、k 近邻法、数据降维、聚类和 EM 算法；第二篇：机器学习高级篇（第 7~10 章），讲述经典而常用的高级机器学习算法，包括支持向量机、人工神经网络、半监督学习和集成学习；第三篇：机器学习工程篇（第 11~12 章），讲述机器学习工程中的实际技术，包括数据预处理，模型评估、选择与验证等；第四篇：Kaggle 实战篇（第 13 章），讲述一个 Kaggle 竞赛题目的实战。

本书内容丰富、深入浅出，算法与代码双管齐下，无论你是新手还是有经验的读者，都能快速学到你想要的知识。本书可供为高等院校计算机、金融、信息、自动化及相关理工科专业的本科生或研究生使用，也可供对机器学习感兴趣的研究人员和工程技术人员阅读参考。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

Python 大战机器学习：数据科学家的第一个小目标 / 华校专，王正林编著.—北京：电子工业出版社，2017.3

（数据科学家成长之路）

ISBN 978-7-121-30894-9

I. ① P…II. ① 华… ② 王…III. ① 软件工具－程序设计 IV. ① TP311.561

中国版本图书馆 CIP 数据核字（2017）第 022099 号

策划编辑：张月萍
责任编辑：徐津平
特约编辑：顾慧芳
印　　刷：北京京科印刷有限公司
装　　订：北京京科印刷有限公司
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×1092　1/16　印张：28　字数：716 千字
版　　次：2017 年 3 月第 1 版
印　　次：2017 年 3 月第 1 次印刷
定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819　faq@phei.com.cn。

自序

本人华校专，性别男，爱好女 ^_^，湖北黄冈人。2004 年我考入清华大学航天学院工程力学系，喜欢读书，在清华的四年我每年的成绩都是本系的 Top 1。在清华园里我养成了记笔记的习惯，大学四年我做了大量的学习笔记，这一良好的习惯一直坚持到现在，十多年的时间我已经记了三千多页的笔记，如图 1 和图 2 所示。

研究生阶段我被免试保送到国防科大计算机学院读研并入伍，研究生毕业之后一直在某部队工作至 2016 年。工作期间我阅读了大量的计算机书籍并编写了大量的代码，从操作系统底层开发到应用 App 开发。这个阶段是我从学生到工程师的一个转变阶段，也是我个人知识体系的建设阶段。对于不理解的内容我反复读、反复研究。记得学习《算法导论》的时候，我阅读了不下四轮，做了两轮笔记，并且仿照 C++ STL 的风格实现了其中的各种算法（算法导论的 C++ 实现我已经放在个人的 github 上）。

我个人比较喜欢研究算法，在这方面我比较有优势。一是我数学能力比较强，作为曾经的清华学霸，我数学相关的课程平均分不低于 95 分（我本科四年的平均学分积不低于 90 分）。另一方面是我编程功底比较强，尤其是精通 C/C++/Python 三门语言。在学习机器学习这个方向的时候，理论方面我结合了斯坦福大学的机器学习课程，李航老师的《统计学习方法》和周志华老师的《机器学习》课程，实践方面我使用 Python 的 scikit-learn 包提供的 API 函数，包里面所涵盖的算法接口非常全面，更令人振奋的是，其用户手册写得非常好，我发现这是一条快捷的学习路径。

机器学习是一门理论与实践结合非常紧密的学科，理论提供了各种算法处理问题的边界，即有的算法适合处理问题 A，不适合问题 B；而另外一些算法适合处理问题 B 不适合处理问题 A。如果不懂得理论，那么对于某个具体问题，你就完全不知道应该采用哪种算法，以及当你采用了某个算法时各类超参数的物理意义。如果没有扎实的写代码实践，那么你可能采用了一个看起来很美好的算法，但是实际操作中因为各种条件不满足，最后要么预测性能很差，要么运行时间能让你崩溃，停留在“看上去很美”的尴尬状态。

2016 年，我顺利从部队退役，一次在北京旅游时，抱着试试的态度，我轻松拿到了阿里的算法 offer，2017 年年初，我在美丽的杭州入职！

看书、笔记、编程这三样是学习机器学习必须的，效果很棒，我已经验证了，该你了！

在机器学习的路上，我们一起同行！

2017 年 1 月，入职日于杭州阿里总部大楼前

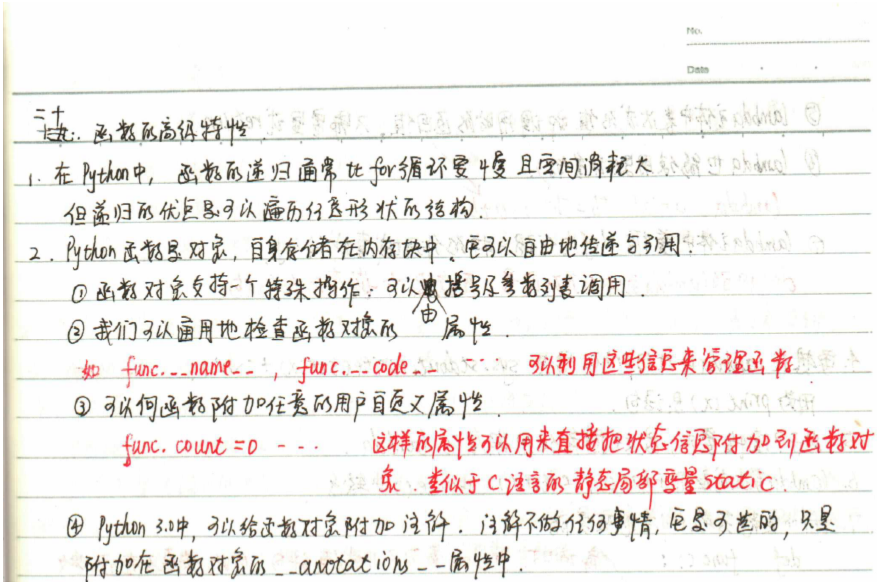


图 1 《Python 学习手册》(中文第 4 版)笔记摘选

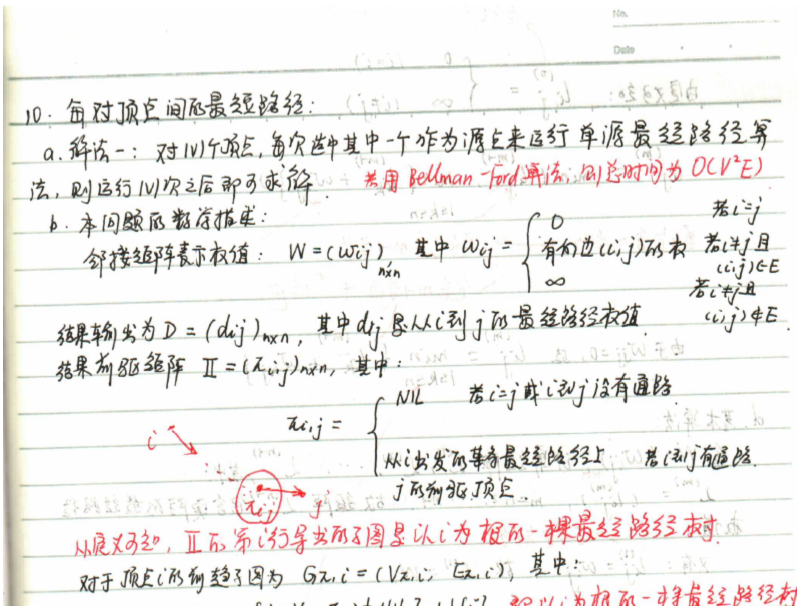


图 2 《算法导论》(中文第 2 版)笔记摘选

前言

拥抱大数据时代

“大家还没搞清 PC 时代的时候，移动互联网来了，还没搞清移动互联网的时候，大数据时代来了。”马云在 2013 年淘宝十周年晚会上的这句话，仿佛一下子拉开了大数据时代的序幕。

新的时代，需要新的技术；新的技术，需要新的人才。全球最著名的管理咨询公司麦肯锡预测“到 2018 年，美国在‘深度分析’人才方面将面临 14 万至 19 万的人才缺口；在‘能够分析数据帮助公司做出商业决策’方面将面临 150 万的人才缺口”。清华大学计算机系教授武永卫 2016 年 5 月透露了一组数据：未来 3~5 年，中国需要 180 万数据人才，但目前只有约 30 万人。

大数据时代，做大数据分析的人有了一个更“性感”的名字，叫做数据科学家（Data Scientist）。《哈佛商业评论》声称，21 世纪最富挑战的工作是数据科学家。时下最热门的职业是数据科学家，而不是传统的信息科学家，也不是大数据工程师。

在数据科学家必备的技能中，机器学习和 Python 应该是位列前五的两项。机器学习炙手可热，在互联网、金融保险、制造业、零售业、医疗等产业领域发挥了越来越大的作用，关注度也越来越高。而 Python 则是最 in 的语言，“人生苦短，我用 Python”^_^

怎么用这本书?

机器学习既有算法又有实现，还是比较高深的，算法太难，啃不动，代码太浅，钻不下去。我们的目标是让您快速上手，在内容组织上我们是动了心思的，采用“原理笔记精华 + 算法 Python 实现 + 问题实例 + 实际代码 + 运行调参”的形式，理论与实践交织着展开，算法原理与编程实战并重。

全书分 13 章进行展开，从内容上分为四篇：机器学习基础篇、机器学习高级篇、机器学习工程篇和 Kaggle 实战篇。

第一篇：机器学习基础篇（第 1~6 章）

包括线性模型、决策树、贝叶斯分类、k 近邻法、数据降维、聚类和 EM 算法等内容。

这些基础算法非常经典，原理也相对简单，是入门的最佳选择，掌握这些算法，才能更好地理解后续的高级算法。非菜鸟可以直接忽略这部分。

第二篇：机器学习高级篇（第 7~10 章）

包括支持向量机、人工神经网络、半监督学习和集成学习等内容。

这些高级算法是目前应用非常广泛，也是效果不错的算法，需要深入理解算法的原理、优劣势等特点以及应用场景，要能达到应用自如的程度。

第三篇：机器学习工程篇（第 11~12 章）

讲述机器学习工程中的实际技术，包括数据预处理，模型评估、选择与验证等内容。

数据清洗、数据预处理和模型评估选择在实际中非常重要，在整个工程项目的开发过程中通常占到一半以上的时间，这部分给出的一些步骤和方法是实践的精华，值得熟练掌握。

第四篇：Kaggle 实战篇（第 13 章）

Step-by-step 讲述一个 Kaggle 竞赛题目的实战，有代码，有分析。

Kaggle 是目前顶级的数据科学比赛平台，很多机器学习的牛人都在这里玩过，咱们可以学习牛人好的算法，也可以启发自己的思路。对于梦想成为牛人的您，还是去里面混混先：）万一拿了个好的名次呢，拿个一流公司的 offer 还是很 easy 的。

本书的代码全部开源，请自行下载`https://github.com/huaxz1986/git_book`，也欢迎在这上面交流。

由于作者水平和经验有限，书中错漏之处在所难免，敬请读者指正，我的电子邮箱是`wa_2003@126.com`。

作者
2017 年元旦于北京

目录

第一篇　机器学习基础篇 1

第 1 章　线性模型 2
1.1　概述 2
1.2　算法笔记精华 2
1.2.1　普通线性回归 2
1.2.2　广义线性模型 5
1.2.3　逻辑回归 5
1.2.4　线性判别分析 7
1.3　Python 实战 10
1.3.1　线性回归模型 11
1.3.2　线性回归模型的正则化 12
1.3.3　逻辑回归 22
1.3.4　线性判别分析 26

第 2 章　决策树 30
2.1　概述 30
2.2　算法笔记精华 30
2.2.1　决策树原理 30
2.2.2　构建决策树的 3 个步骤 31
2.2.3　CART 算法 37
2.2.4　连续值和缺失值的处理 42
2.3　Python 实战 43
2.3.1　回归决策树（DecisionTreeRegressor） 43
2.3.2　分类决策树（DecisionTreeClassifier） 49
2.3.3　决策图 54

第 3 章　贝叶斯分类器 55
3.1　概述 55

3.2 算法笔记精华 55
3.2.1 贝叶斯定理 55
3.2.2 朴素贝叶斯法 56
3.3 Python 实战 59
3.3.1 高斯贝叶斯分类器（GaussianNB） 61
3.3.2 多项式贝叶斯分类器（MultinomialNB） 62
3.3.3 伯努利贝叶斯分类器（BernoulliNB） 65
3.3.4 递增式学习 partial_fit 方法 69

第 4 章 k 近邻法 70
4.1 概述 70
4.2 算法笔记精华 70
4.2.1 kNN 三要素 70
4.2.2 k 近邻算法 72
4.2.3 kd 树 73
4.3 Python 实践 74

第 5 章 数据降维 83
5.1 概述 83
5.2 算法笔记精华 83
5.2.1 维度灾难与降维 83
5.2.2 主成分分析（PCA） 84
5.2.3 SVD 降维 91
5.2.4 核化线性（KPCA）降维 91
5.2.5 流形学习降维 93
5.2.6 多维缩放（MDS）降维 93
5.2.7 等度量映射（Isomap）降维 96
5.2.8 局部线性嵌入（LLE） 97
5.3 Python 实战 99
5.4 小结 118

第 6 章 聚类和 EM 算法 119
6.1 概述 119
6.2 算法笔记精华 120
6.2.1 聚类的有效性指标 120
6.2.2 距离度量 122
6.2.3 原型聚类 123
6.2.4 密度聚类 126

6.2.5 层次聚类 127
6.2.6 EM 算法 128
6.2.7 实际中的聚类要求 136
6.3 Python 实战 137
6.3.1 K 均值聚类（KMeans） 138
6.3.2 密度聚类（DBSCAN） 143
6.3.3 层次聚类（AgglomerativeClustering） 146
6.3.4 混合高斯（GaussianMixture）模型 149
6.4 小结 153

第二篇 机器学习高级篇 155

第 7 章 支持向量机 156

7.1 概述 156
7.2 算法笔记精华 157
7.2.1 线性可分支持向量机 157
7.2.2 线性支持向量机 162
7.2.3 非线性支持向量机 166
7.2.4 支持向量回归 167
7.2.5 SVM 的优缺点 170
7.3 Python 实战 170
7.3.1 线性分类 SVM 171
7.3.2 非线性分类 SVM 175
7.3.3 线性回归 SVR 182
7.3.4 非线性回归 SVR 186

第 8 章 人工神经网络 192

8.1 概述 192
8.2 算法笔记精华 192
8.2.1 感知机模型 192
8.2.2 感知机学习算法 194
8.2.3 神经网络 197
8.3 Python 实战 205
8.3.1 感知机学习算法的原始形式 205
8.3.2 感知机学习算法的对偶形式 209
8.3.3 学习率与收敛速度 212
8.3.4 感知机与线性不可分数据集 213
8.3.5 多层神经网络 215

8.3.6 多层神经网络与线性不可分数据集 216
8.3.7 多层神经网络的应用 219

第 9 章 半监督学习 225
9.1 概述 225
9.2 算法笔记精华 226
9.2.1 生成式半监督学习方法 226
9.2.2 图半监督学习 228
9.3 Python 实战 234
9.4 小结 243

第 10 章 集成学习 244
10.1 概述 244
10.2 算法笔记精华 244
10.2.1 集成学习的原理及误差 244
10.2.2 Boosting 算法 246
10.2.3 AdaBoost 算法 246
10.2.4 AdaBoost 与加法模型 252
10.2.5 提升树 253
10.2.6 Bagging 算法 256
10.2.7 误差-分歧分解 257
10.2.8 多样性增强 259
10.3 Python 实战 260
10.3.1 AdaBoost 261
10.3.2 Gradient Tree Boosting 272
10.3.3 Random Forest 288
10.4 小结 298

第三篇 机器学习工程篇 299

第 11 章 数据预处理 300
11.1 概述 300
11.2 算法笔记精华 300
11.2.1 去除唯一属性 300
11.2.2 处理缺失值的三种方法 301
11.2.3 常见的缺失值补全方法 302
11.2.4 特征编码 307

11.2.5 数据标准化、正则化 308
11.2.6 特征选择 310
11.2.7 稀疏表示和字典学习 313
11.3 Python 实践 316
11.3.1 二元化 316
11.3.2 独热码 317
11.3.3 标准化 321
11.3.4 正则化 325
11.3.5 过滤式特征选取 326
11.3.6 包裹式特征选取 330
11.3.7 嵌入式特征选取 334
11.3.8 学习器流水线（Pipeline） 339
11.3.9 字典学习 340

第 12 章 模型评估、选择与验证 345
12.1 概述 345
12.2 算法笔记精华 346
12.2.1 损失函数和风险函数 346
12.2.2 模型评估方法 348
12.2.3 模型评估 349
12.2.4 性能度量 350
12.2.5 偏差方差分解 356
12.3 Python 实践 357
12.3.1 损失函数 357
12.3.2 数据集切分 359
12.3.3 性能度量 370
12.3.4 参数优化 387

第四篇 Kaggle 实战篇 401

第 13 章 Kaggle 牛刀小试 402
13.1 Kaggle 简介 402
13.2 清洗数据 403
13.2.1 加载数据 403
13.2.2 合并数据 406
13.2.3 拆分数据 407
13.2.4 去除唯一值 408
13.2.5 数据类型转换 410

13.2.6 Data_Cleaner 类 412
13.3 数据预处理 415
13.3.1 独热码编码 415
13.3.2 归一化处理 419
13.3.3 Data_Preprocesser 类 421
13.4 学习曲线和验证曲线 424
13.4.1 程序说明 424
13.4.2 运行结果 430
13.5 参数优化 433
13.6 小结 435

全书符号 436

第一篇

机器学习基础篇

第 1 章

线性模型

1.1 概述

给定样本 $\vec{\boldsymbol{x}}$，我们用列向量表示该样本 $\vec{\boldsymbol{x}} = (x^{(1)}, x^{(2)}, \cdots, x^{(n)})^T$。样本有 n 种特征，我们用 $x^{(i)}$ 表示样本 $\vec{\boldsymbol{x}}$ 的第 i 个特征。线性模型 (linear model) 的形式为：

$$f(\vec{\boldsymbol{x}}) = \vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} + b$$

其中 $\vec{\boldsymbol{w}} = (w^{(1)}, w^{(2)}, \cdots, w^{(n)})^T$ 为每个特征对应的权重生成的权重向量，称为权重向量，权重向量直观地表达了各个特征在预测中的重要性。

线性模型中的“线性”其实就是一系列一次特征的线性组合，在二维空间中是一条直线，在三维空间中是一个平面，然后推广到 n 维空间，这样可以理解为广义线性模型。

线性模型非常简单，易于建模，应用广泛，它还有多种推广形式，常见的有广义线性模型，包括岭回归、lasso 回归、Elastic Net、逻辑回归、线性判别分析等。本章将介绍这些模型的基本思想、优缺点以及如何用 Python 实现。

1.2 算法笔记精华

1.2.1 普通线性回归

线性回归是一种回归分析技术，回归分析本质上就是一个函数估计的问题（函数估计包括参数估计和非参数估计两类），就是找出因变量和自变量之间的因果关系。回归分析的因变量应该是连续变量，若因变量为离散变量，则问题转化为分类问题，回归分析是一个有监督学习的问题。

给定数据集 $T=\{(\vec{\boldsymbol{x}}_1,y_1),(\vec{\boldsymbol{x}}_2,y_2),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\},\vec{\boldsymbol{x}}_i\in\mathcal{X}\subseteq\mathbb{R}^n,y_i\in\mathcal{Y}\subseteq\mathbb{R},i=1,2,\cdots,N$，其中 $\vec{\boldsymbol{x}}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(n)})^T$。我们需要学习的模型为：

$$f(\vec{\boldsymbol{x}})=\vec{\boldsymbol{w}}\cdot\vec{\boldsymbol{x}}+b$$

也即：根据已知的数据集 T 来计算参数 $\vec{\boldsymbol{w}}$ 和 b。

对于给定的样本 $\vec{\boldsymbol{x}}_i$，其预测值为 $\hat{y}_i=f(\vec{\boldsymbol{x}}_i)=\vec{\boldsymbol{w}}\cdot\vec{\boldsymbol{x}}_i+b$。我们采用平方损失函数，则在训练集 T 上，模型的损失函数为：

$$L(f)=\sum_{i=1}^{N}\left(\hat{y}_i-y_i\right)^2=\sum_{i=1}^{N}\left(\vec{\boldsymbol{w}}\cdot\vec{\boldsymbol{x}}_i+b-y_i\right)^2$$

我们的目标是损失函数最小化，即：

$$(\vec{\boldsymbol{w}}^*,b^*)=\arg\min_{\vec{\boldsymbol{w}},b}\sum_{i=1}^{N}\left(\vec{\boldsymbol{w}}\cdot\vec{\boldsymbol{x}}_i+b-y_i\right)^2$$

可以用梯度下降法来求解上述最优化问题的数值解。在使用梯度下降法时，要注意特征归一化（Feature Scaling），这也是许多机器学习模型都需要注意的问题，这么重要的问题，我们一定要讲三遍！

特征归一化有两个好处。(1) 提升模型的收敛速度，比如两个特征 x_1 和 x_2，x_1 的取值为 0~2000，而 x_2 的取值为 1~5，假如只有这两个特征，对其进行优化时，会得到一个窄长的椭圆形，导致在梯度下降时，梯度的方向为垂直等高线的方向而走之字形路线，这样会使迭代很慢。相比之下，归一化之后，是一个圆形，梯度的方向为直接指向圆心，迭代就会很快。可见，归一化可以大大减少寻找最优解的时间。(2) 提升模型精度，归一化的另一好处是提高精度，这在涉及一些距离计算的算法时效果显著，比如算法要计算欧氏距离，上面 x_2 的取值范围比较小，涉及距离计算时其对结果的影响远比 x_1 带来的小，所以这就会造成精度的损失。所以归一化很有必要，它可以让各个特征对结果做出的贡献相同。在求解线性回归的模型时，还有一个问题要注意，那就是特征组合问题，比如房子的长度和宽度作为两个特征参与模型的构造，不如把其相乘得到面积作为一个特征来进行求解，这样在特征选择上就做了减少维度的工作。

回过头来，上述最优化问题实际上是有解析解的，可以用最小二乘法求解解析解，该问题称为多元线性回归 (`multivariate linear regression`)。

令：

$$\vec{\tilde{\boldsymbol{w}}}=(w^{(1)},w^{(2)},\cdots,w^{(n)},b)^T=(\vec{\boldsymbol{w}}^T,b)^T$$
$$\vec{\tilde{\boldsymbol{x}}}=(x^{(1)},x^{(2)},\cdots,x^{(n)},1)^T=(\vec{\boldsymbol{x}}^T,1)^T$$
$$\vec{\boldsymbol{y}}=(y_1,y_2,\cdots,y_N)^T$$

则有：

$$\sum_{i=1}^{N}\left(\vec{\boldsymbol{w}}\cdot\vec{\boldsymbol{x}}_i+b-y_i\right)^2=\left(\vec{\boldsymbol{y}}-(\vec{\tilde{\boldsymbol{x}}}_1,\vec{\tilde{\boldsymbol{x}}}_2,\cdots,\vec{\tilde{\boldsymbol{x}}}_N)^T\vec{\tilde{\boldsymbol{w}}}\right)^T\left(\vec{\boldsymbol{y}}-(\vec{\tilde{\boldsymbol{x}}}_1,\vec{\tilde{\boldsymbol{x}}}_2,\cdots,\vec{\tilde{\boldsymbol{x}}}_N)^T\vec{\tilde{\boldsymbol{w}}}\right)$$

令：

$$\vec{\boldsymbol{x}}=(\vec{\tilde{\boldsymbol{x}}}_1,\vec{\tilde{\boldsymbol{x}}}_2,\cdots,\vec{\tilde{\boldsymbol{x}}}_N)^T=\begin{bmatrix}\vec{\tilde{\boldsymbol{x}}}_1^T\\\vec{\tilde{\boldsymbol{x}}}_2^T\\\vdots\\\vec{\tilde{\boldsymbol{x}}}_N^T\end{bmatrix}=\begin{bmatrix}x_1^{(1)}&x_1^{(2)}&\cdots&x_1^{(n)}&1\\x_2^{(1)}&x_2^{(2)}&\cdots&x_2^{(n)}&1\\\vdots&\vdots&\ddots&\vdots&1\\x_N^{(1)}&x_N^{(2)}&\cdots&x_N^{(n)}&1\end{bmatrix}$$

则：

$$\vec{\tilde{\boldsymbol{w}}}^*=\arg\min_{\vec{\tilde{\boldsymbol{w}}}}(\vec{\boldsymbol{y}}-\vec{\boldsymbol{x}}\vec{\tilde{\boldsymbol{w}}})^T(\vec{\boldsymbol{y}}-\vec{\boldsymbol{x}}\vec{\tilde{\boldsymbol{w}}})$$

令 $E_{\vec{\tilde{\boldsymbol{w}}}}=(\vec{\boldsymbol{y}}-\vec{\boldsymbol{x}}\vec{\tilde{\boldsymbol{w}}})^T(\vec{\boldsymbol{y}}-\vec{\boldsymbol{x}}\vec{\tilde{\boldsymbol{w}}})$，求它的极小值。对 $\vec{\tilde{\boldsymbol{w}}}$ 求导令导数为零，得到解析解：

$$\frac{\partial E_{\vec{\tilde{\boldsymbol{w}}}}}{\partial\vec{\tilde{\boldsymbol{w}}}}=2\vec{\boldsymbol{x}}^T(\vec{\boldsymbol{x}}\vec{\tilde{\boldsymbol{w}}}-\vec{\boldsymbol{y}})=\vec{\boldsymbol{0}}\Longrightarrow\vec{\boldsymbol{x}}^T\vec{\boldsymbol{x}}\vec{\tilde{\boldsymbol{w}}}=\vec{\boldsymbol{x}}^T\vec{\boldsymbol{y}}$$

- 当 $\vec{\boldsymbol{x}}^T\vec{\boldsymbol{x}}$ 为满秩矩阵或者正定矩阵时，可得：

$$\vec{\tilde{\boldsymbol{w}}}^*=(\vec{\boldsymbol{x}}^T\vec{\boldsymbol{x}})^{-1}\vec{\boldsymbol{x}}^T\vec{\boldsymbol{y}}$$

  其中 $(\vec{\boldsymbol{x}}^T\vec{\boldsymbol{x}})^{-1}$ 为 $\vec{\boldsymbol{x}}^T\vec{\boldsymbol{x}}$ 的逆矩阵。于是学得的多元线性回归模型为：

$$f(\vec{\tilde{\boldsymbol{x}}}_i)=\vec{\tilde{\boldsymbol{x}}}_i^T\vec{\tilde{\boldsymbol{w}}}^*$$

- 当 $\vec{\boldsymbol{x}}^T\vec{\boldsymbol{x}}$ 不是满秩矩阵时。比如 $N<n$（样本数量小于特征种类的数量），根据 $\vec{\boldsymbol{x}}$ 的秩小于等于 (N,n) 中的最小值，即小于等于 N（矩阵的秩一定小于等于矩阵的行数和列数）；而矩阵 $\vec{\boldsymbol{x}}^T\vec{\boldsymbol{x}}$ 是 $n\times n$ 大小的，它的秩一定小于等于 N，因此不是满秩矩阵。此时存在多个解析解。常见的做法是引入正则化项，如 L_1 正则化或者 L_2 正则化。以 L_2 正则化为例：

$$\vec{\tilde{\boldsymbol{w}}}^*=\arg\min_{\vec{\tilde{\boldsymbol{w}}}}\left[(\vec{\boldsymbol{y}}-\vec{\boldsymbol{x}}\vec{\tilde{\boldsymbol{w}}})^T(\vec{\boldsymbol{y}}-\vec{\boldsymbol{x}}\vec{\tilde{\boldsymbol{w}}})+\lambda||\vec{\tilde{\boldsymbol{w}}}||_2^2\right]$$

  其中，$\lambda>0$ 调整正则化项与均方误差的比例；$||...||_2$ 为 L_2 范数。

根据上述原理，我们得到多元线性回归算法：

- 输入：数据集 $T=\{(\vec{\boldsymbol{x}}_1,y_1),(\vec{\boldsymbol{x}}_2,y_2),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\},\vec{\boldsymbol{x}}_i\in\mathcal{X}\subseteq\mathbb{R}^n,y_i\in\mathcal{Y}\subseteq\mathbb{R},i=1,2,\cdots,N$，正则化项系数 $\lambda>0$。

❑ 输出：

$$f(\vec{x}) = \vec{w} \cdot \vec{x} + b$$

❑ 算法步骤：

❍ 令：

$$\vec{\tilde{w}} = (w^{(1)}, w^{(2)}, \cdots, w^{(n)}, b)^T = (\vec{w}^T, b)^T$$
$$\vec{\tilde{x}} = (x^{(1)}, x^{(2)}, \cdots, x^{(n)}, 1)^T = (\vec{x}^T, 1)^T$$
$$\vec{y} = (y_1, y_2, \cdots, y_N)^T$$

计算

$$\mathbf{x} = (\vec{\tilde{x}}_1, \vec{\tilde{x}}_2, \cdots, \vec{\tilde{x}}_N)^T = \begin{bmatrix} \vec{\tilde{x}}_1^T \\ \vec{\tilde{x}}_2^T \\ \vdots \\ \vec{\tilde{x}}_N^T \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(n)} & 1 \\ x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(n)} & 1 \\ \vdots & \vdots & \ddots & \vdots & 1 \\ x_N^{(1)} & x_N^{(2)} & \cdots & x_N^{(n)} & 1 \end{bmatrix}$$

❍ 求解：

$$\vec{\tilde{w}}^* = \arg\min_{\vec{\tilde{w}}} \left[(\vec{y} - \mathbf{x}\vec{\tilde{w}})^T (\vec{y} - \mathbf{x}\vec{\tilde{w}}) + \lambda ||\vec{\tilde{w}}||_2^2 \right]$$

❍ 最终学得模型：

$$f(\vec{\tilde{x}}_i) = \vec{\tilde{x}}_i^T \vec{\tilde{w}}^*$$

1.2.2 广义线性模型

考虑单调可导函数 $h(\cdot)$，令 $h(y) = \vec{w}^T\vec{x} + b$ ，这样得到的模型称为广义线性模型（`generalized linear model`）。

广义线性模型的一个典型的例子就是对数线性回归。当 $h(\cdot) = \ln(\cdot)$ 时的广义线性模型就是对数线性回归，即

$$\ln y = \vec{w}^T\vec{x} + b$$

它是通过 $\exp(\vec{w}^T\vec{x} + b)$ 拟合 y 的。它虽然称为广义线性回归，但实质上是非线性的。

1.2.3 逻辑回归

前述的学习方法都是使用线性模型进行回归学习，而线性模型也可以用于分类。考虑二类分类问题，给定数据集 $T = \{(\vec{x}_1, y_1), (\vec{x}_2, y_2), \cdots, (\vec{x}_N, y_N)\}, \vec{x}_i \in \mathcal{X} \subseteq \mathbb{R}^n, y_i \in \mathcal{Y} = \{0, 1\}, i = 1, 2, \cdots, N$，其中 $\vec{x}_i = (x_i^{(1)}, x_i^{(2)}, \cdots, x_i^{(n)})^T$。我们需要知道 $P(y/\vec{x})$，这里用条件概率的原因是：预测的时候都是已知 $\vec{x}$ ，然后需要判断此时对应的 y 值。

考虑到 $\vec{w} \cdot \vec{x} + b$ 取值是连续的，因此它不能拟合离散变量。可以考虑用它来拟合条件概率 $P(y = 1/\vec{x})$，因为概率的取值也是连续的。但是对于 $\vec{w} \neq \vec{0}$（若等于零向量则没有求解

的价值），$\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} + b$ 取值是从 $-\infty \sim +\infty$，不符合概率取值为 $0 \sim 1$，因此考虑采用广义线性模型，最理想的是单位阶跃函数：

$$P(y=1/\vec{\boldsymbol{x}}) = \begin{cases} 0, & z < 0 \\ 0.5, & z = 0 \\ 1, & z > 0 \end{cases} \quad , z = \vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} + b$$

但是阶跃函数不满足单调可导的性质。退而求其次，我们寻找一个可导的、与阶跃函数相似的函数。对数概率函数 (`logistic function`) 就是这样的一个替代函数：

$$P(y=1/\vec{\boldsymbol{x}}) = \frac{1}{1-e^{-z}} \quad , z = \vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} + b$$

由于 $P(y=0/\vec{\boldsymbol{x}}) = 1 - P(y=1/\vec{\boldsymbol{x}})$，则有：$\ln \frac{P(y=1/\vec{\boldsymbol{x}})}{P(y=0/\vec{\boldsymbol{x}})} = z = \vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} + b$。比值 $\frac{P(y=1/\vec{\boldsymbol{x}})}{P(y=0/\vec{\boldsymbol{x}})}$ 表示样本为正例的可能性比反例的可能性，称为概率 (`odds`)，反映了样本作为正例的相对可能性。概率的对数称为对数概率 (`log odds`，也称为`logit`)。

下面给出逻辑回归模型参数估计：给定训练数据集 $T = \{(\vec{\boldsymbol{x}}_1, y_1), (\vec{\boldsymbol{x}}_2, y_2), \cdots, (\vec{\boldsymbol{x}}_N, y_N)\}$，其中 $\vec{\boldsymbol{x}}_i \in \mathbb{R}^n, y_i \in \{0, 1\}$。模型估计的原理是：用极大似然估计法估计模型参数。

为了便于讨论，我们将参数 b 吸收进 $\vec{\boldsymbol{w}}$ 中，令：

$$\vec{\tilde{\boldsymbol{w}}} = (w^{(1)}, w^{(2)}, \cdots, w^{(n)}, b)^T \in \mathbb{R}^{n+1}, \quad \vec{\tilde{\boldsymbol{x}}} = (x^{(1)}, x^{(2)} \cdots, x^{(n)}, 1)^T \in \mathbb{R}^{n+1}$$

令 $P(Y=1/\vec{\tilde{\boldsymbol{x}}}) = \pi(\vec{\tilde{\boldsymbol{x}}}) = \frac{\exp(\vec{\tilde{\boldsymbol{w}}} \cdot \vec{\tilde{\boldsymbol{x}}})}{1+\exp(\vec{\tilde{\boldsymbol{w}}} \cdot \vec{\tilde{\boldsymbol{x}}})}, P(Y=0/\vec{\tilde{\boldsymbol{x}}}) = 1 - \pi(\vec{\tilde{\boldsymbol{x}}})$，则似然函数为：

$$\prod_{i=1}^{N} [\pi(\vec{\tilde{\boldsymbol{x}}}_i)]^{y_i} [1 - \pi(\vec{\tilde{\boldsymbol{x}}}_i)]^{1-y_i}$$

对数似然函数为：

$$\begin{aligned} L(\vec{\tilde{\boldsymbol{w}}}) &= \sum_{i=1}^{N} [y_i \log \pi(\vec{\tilde{\boldsymbol{x}}}_i) + (1-y_i) \log(1 - \pi(\vec{\tilde{\boldsymbol{x}}}_i))] \\ &= \sum_{i=1}^{N} [y_i \log \frac{\pi(\vec{\tilde{\boldsymbol{x}}}_i)}{1 - \pi(\vec{\tilde{\boldsymbol{x}}}_i)} + \log(1 - \pi(\vec{\tilde{\boldsymbol{x}}}_i))] \end{aligned}$$

又由于 $\pi(\vec{\tilde{\boldsymbol{x}}}) = \frac{\exp(\vec{\tilde{\boldsymbol{w}}} \cdot \vec{\tilde{\boldsymbol{x}}})}{1+\exp(\vec{\tilde{\boldsymbol{w}}} \cdot \vec{\tilde{\boldsymbol{x}}})}$，因此：

$$L(\vec{\tilde{\boldsymbol{w}}}) = \sum_{i=1}^{N} [y_i (\vec{\tilde{\boldsymbol{w}}} \cdot \vec{\tilde{\boldsymbol{x}}}_i) - \log(1 + \exp(\vec{\tilde{\boldsymbol{w}}} \cdot \vec{\tilde{\boldsymbol{x}}}_i))]$$

对 $L(\vec{\boldsymbol{w}})$ 求极大值，得到 $\vec{\boldsymbol{w}}$ 的估计值。设估计值为 $\vec{\boldsymbol{w}}^*$，则逻辑回归模型为：

$$P(Y=1/X=\vec{\tilde{\boldsymbol{x}}})=\frac{\exp(\vec{\tilde{\boldsymbol{w}}}^*\cdot\vec{\tilde{\boldsymbol{x}}})}{1+\exp(\vec{\tilde{\boldsymbol{w}}}^*\cdot\vec{\tilde{\boldsymbol{x}}})}$$
$$P(Y=0/X=\vec{\tilde{\boldsymbol{x}}})=\frac{1}{1+\exp(\vec{\tilde{\boldsymbol{w}}}^*\cdot\vec{\tilde{\boldsymbol{x}}})}$$

> 通常用梯度下降法或者拟牛顿法来求解该最大值问题。

以上讨论的都是二类分类的逻辑回归模型，可以推广到多类分类逻辑回归模型。设离散型随机变量 Y 的取值集合为：$\{1,2,\cdots,K\}$，则多类分类逻辑回归模型为：

$$P(Y=k/\vec{\tilde{\boldsymbol{x}}})=\frac{\exp(\vec{\tilde{\boldsymbol{w}}}_k\cdot\vec{\tilde{\boldsymbol{x}}})}{1+\sum_{k=1}^{K-1}\exp(\vec{\tilde{\boldsymbol{w}}}_k\cdot\vec{\tilde{\boldsymbol{x}}})},k=1,2,\cdots,K-1$$
$$P(Y=K/\vec{\tilde{\boldsymbol{x}}})=\frac{1}{1+\sum_{k=1}^{K-1}\exp(\vec{\tilde{\boldsymbol{w}}}_k\cdot\vec{\tilde{\boldsymbol{x}}})},\vec{\tilde{\boldsymbol{x}}}\in\mathbb{R}^{n+1},\vec{\tilde{\boldsymbol{w}}}_k\in\mathbb{R}^{n+1}$$

其参数估计方法类似二类分类逻辑回归模型。

1.2.4 线性判别分析

线性判别分析（`Linear Discriminant Analysis，LDA`）的思想是：

- 训练时：设法将训练样本投影到一条直线上，使得同类样本的投影点尽可能地接近、异类样本的投影点尽可能地远离。要学习的就是这样的一条直线。
- 预测时：将待预测样本投影到学到的直线上，根据它的投影点的位置来判定它的类别。

考虑二类分类问题，给定数据集 $T=\{(\vec{\boldsymbol{x}}_1,y_1),(\vec{\boldsymbol{x}}_2,y_2),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\},\vec{\boldsymbol{x}}_i\in\mathcal{X}\subseteq\mathbb{R}^n,y_i\in\mathcal{Y}=\{0,1\},i=1,2,\cdots,N$，其中 $\vec{\boldsymbol{x}}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(n)})^T$ 。

- 设 T_0 表示类别为 0 的样例的集合，这些样例的均值向量为 $\vec{\mu}_0=(\mu_0^{(1)},\mu_0^{(2)},\cdots,\mu_0^{(n)})^T$，这些样例的特征之间协方差矩阵为 Σ_0（协方差矩阵大小为 $n\times n$）。
- 设 T_1 表示类别为 1 的样例的集合，这些样例的均值向量为 $\vec{\mu}_1=(\mu_1^{(1)},\mu_1^{(2)},\cdots,\mu_1^{(n)})^T$，这些样例的特征之间协方差矩阵为 Σ_1（协方差矩阵大小为 $n\times n$）。

假定直线为 $y=\vec{\boldsymbol{w}}^T\vec{\boldsymbol{x}}$（这里省略了常量 b，因为考察的是样本点在直线上的投影，总可以平行移动直线到原点而保持投影不变，此时 $b=0$ ），其中 $\vec{\boldsymbol{w}}=(w^{(1)},w^{(2)},\cdots,w^{(n)})^T,\vec{\boldsymbol{x}}=(x^{(1)},x^{(2)},\cdots,x^{(n)})^T$。两类样本的线性判别分析如图 1.1 所示。

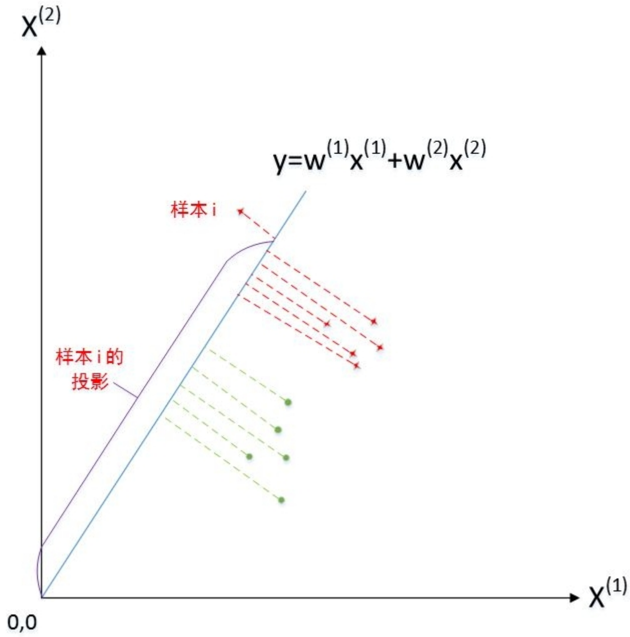


图 1.1 LDA

将数据投影到直线上，则：

- 两类样本的中心在直线上的投影分别为 $\vec{\boldsymbol{w}}^T\vec{\mu}_0$ 和 $\vec{\boldsymbol{w}}^T\vec{\mu}_1$。
- 两类样本投影的方差分别为 $\vec{\boldsymbol{w}}^T\Sigma_0\vec{\boldsymbol{w}}$ 和 $\vec{\boldsymbol{w}}^T\Sigma_1\vec{\boldsymbol{w}}$。

我们的目标是：同类样本的投影点尽可能地接近、异类样本的投影点尽可能地远离。那么可以使同类样例投影点的方差尽可能地小，即 $\vec{\boldsymbol{w}}^T\Sigma_0\vec{\boldsymbol{w}}+\vec{\boldsymbol{w}}^T\Sigma_1\vec{\boldsymbol{w}}$ 尽可能地小；可以使异类样例的中心的投影点尽可能地远，即 $||\vec{\boldsymbol{w}}^T\vec{\mu}_0-\vec{\boldsymbol{w}}^T\vec{\mu}_1||_2^2$ 尽可能地大。于是得到最大化的目标：

$$J=\frac{||\vec{\boldsymbol{w}}^T\vec{\mu}_0-\vec{\boldsymbol{w}}^T\vec{\mu}_1||_2^2}{\vec{\boldsymbol{w}}^T\Sigma_0\vec{\boldsymbol{w}}+\vec{\boldsymbol{w}}^T\Sigma_1\vec{\boldsymbol{w}}}=\frac{\vec{\boldsymbol{w}}^T(\vec{\mu}_0-\vec{\mu}_1)(\vec{\mu}_0-\vec{\mu}_1)^T\vec{\boldsymbol{w}}}{\vec{\boldsymbol{w}}^T(\Sigma_0+\Sigma_1)\vec{\boldsymbol{w}}}$$

定义类内散度矩阵 `within-class scatter matrix`：

$$\boldsymbol{S}_w=\Sigma_0+\Sigma_1=\sum_{\vec{\boldsymbol{x}}\in T_0}(\vec{\boldsymbol{x}}-\vec{\mu}_0)(\vec{\boldsymbol{x}}-\vec{\mu}_0)^T+\sum_{\vec{\boldsymbol{x}}\in T_1}(\vec{\boldsymbol{x}}-\vec{\mu}_1)(\vec{\boldsymbol{x}}-\vec{\mu}_1)^T$$

定义类间散度矩阵 `between-class scatter matrix`：$\boldsymbol{S}_b=(\vec{\mu}_0-\vec{\mu}_1)(\vec{\mu}_0-\vec{\mu}_1)^T$，它是向量 $(\vec{\mu}_0-\vec{\mu}_1)$ 与它自身的外积，则LDA最大化的目标为：

$$J=\frac{\vec{\boldsymbol{w}}^T\boldsymbol{S}_b\vec{\boldsymbol{w}}}{\vec{\boldsymbol{w}}^T\boldsymbol{S}_w\vec{\boldsymbol{w}}}$$

J 也称为 $\boldsymbol{S}_b$ 与 $\boldsymbol{S}_w$ 的广义瑞利商。现在求解最优化问题：

$$\vec{\boldsymbol{w}}^* = \arg\max_{\vec{\boldsymbol{w}}} \frac{\vec{\boldsymbol{w}}^T \boldsymbol{S}_b \vec{\boldsymbol{w}}}{\vec{\boldsymbol{w}}^T \boldsymbol{S}_w \vec{\boldsymbol{w}}}$$

由于分子与分母都是关于 $\vec{\boldsymbol{w}}$ 的二次项，因此上式的解与 $\vec{\boldsymbol{w}}$ 的长度无关。令 $\vec{\boldsymbol{w}}^T \boldsymbol{S}_w \vec{\boldsymbol{w}} = 1$，则最优化问题改写为：

$$\vec{\boldsymbol{w}}^* = \arg\min_{\vec{\boldsymbol{w}}} -\vec{\boldsymbol{w}}^T \boldsymbol{S}_b \vec{\boldsymbol{w}}$$
$$s.t. \vec{\boldsymbol{w}}^T \boldsymbol{S}_w \vec{\boldsymbol{w}} = 1$$

应用拉格朗日乘子法：

$$\boldsymbol{S}_b \vec{\boldsymbol{w}} = \lambda \boldsymbol{S}_w \vec{\boldsymbol{w}}$$

令 $(\vec{\mu}_0 - \vec{\mu}_1)^T \vec{\boldsymbol{w}} = \lambda_{\vec{\boldsymbol{w}}}$，其中 $\lambda_{\vec{\boldsymbol{w}}}$ 为实数。则 $\boldsymbol{S}_b \vec{\boldsymbol{w}} = (\vec{\mu}_0 - \vec{\mu}_1)(\vec{\mu}_0 - \vec{\mu}_1)^T \vec{\boldsymbol{w}} = \lambda_{\vec{\boldsymbol{w}}}(\vec{\mu}_0 - \vec{\mu}_1)$。$\boldsymbol{S}_b \vec{\boldsymbol{w}} = \lambda_{\vec{\boldsymbol{w}}}(\vec{\mu}_0 - \vec{\mu}_1) = \lambda \boldsymbol{S}_w \vec{\boldsymbol{w}}$。由于与 $\vec{\boldsymbol{w}}$ 的长度无关，可以令 $\lambda_{\vec{\boldsymbol{w}}} = \lambda$，则有：

$$(\vec{\mu}_0 - \vec{\mu}_1) = \boldsymbol{S}_w \vec{\boldsymbol{w}} \Longrightarrow \vec{\boldsymbol{w}} = \boldsymbol{S}_w^{-1} (\vec{\mu}_0 - \vec{\mu}_1)$$

上述讨论的是二类分类LDA算法。可以将它推广到多分类任务中：假定存在 M 个类，属于第 i 个类的样本的集合为 T_i，T_i 中的样例数为 m_i，则有：$\sum_{i=1}^{M} m_i = N$ ，其中 N 为样本总数。设 T_i 表示类别为 i， $i = 1, 2, \cdots, M$ 的样例的集合，这些样例的均值向量为：

$$\vec{\mu}_i = (\mu_i^{(1)}, \mu_i^{(2)}, \cdots, \mu_i^{(n)})^T = \frac{1}{m_i} \sum_{\vec{\boldsymbol{x}}_i \in T_i} \vec{\boldsymbol{x}}_i$$

这些样例的特征之间协方差矩阵为 Σ_i（协方差矩阵大小为 $n \times n$）。定义 $\vec{\mu} = (\mu^{(1)}, \mu^{(2)}, \cdots, \mu^{(n)})^T = \frac{1}{N} \sum_{i=1}^{N} \vec{\boldsymbol{x}}_i$ 是所有样例的均值向量。

- 要使得同类样例的投影点尽可能地接近，则可以使同类样例投影点的方差尽可能地小，因此定义类别的类内散度矩阵为 $\boldsymbol{S}_{wi} = \sum_{\vec{\boldsymbol{x}} \in T_i} (\vec{\boldsymbol{x}} - \vec{\mu}_i)(\vec{\boldsymbol{x}} - \vec{\mu}_i)^T$；定义类内散度矩阵为 $\boldsymbol{S}_w = \sum_{i=1}^{M} \boldsymbol{S}_{wi}$。

> 类别的类内散度矩阵为 $\boldsymbol{S}_{wi} = \sum_{\vec{\boldsymbol{x}} \in T_i} (\vec{\boldsymbol{x}} - \vec{\mu}_i)(\vec{\boldsymbol{x}} - \vec{\mu}_i)^T$，实际上就等于样本集 T_i 的协方差矩阵 Σ_i。

- 要使异类样例的投影点尽可能地远，则可以使异类样例中心的投影点尽可能地远，由于这里有不止两个中心点，因此不能简单地套用二类LDA的做法（即两个中心点的距离）。这里用每一类样本集和的中心点距总的中心点的距离作为度量。考虑到每一类样本集的大小可能不同（密度分布不均），故我们对这个距离加以权重，因此定义类间散度矩阵 $\boldsymbol{S}_b = \sum_{i=1}^{M} m_i (\vec{\mu}_i - \vec{\mu})(\vec{\mu}_i - \vec{\mu})^T$。

> $(\vec{\mu}_i - \vec{\mu})(\vec{\mu}_i - \vec{\mu})^T$ 也是一个协方差矩阵，它刻画的是第 i 类与总体之间的关系。

设 $\boldsymbol{W} \in \mathbb{R}^{n\times(M-1)}$ 是投影矩阵。经过推导可以得到最大化的目标：

$$J = \frac{tr(\boldsymbol{W}^T\boldsymbol{S}_b\boldsymbol{W})}{tr(\boldsymbol{W}^T\boldsymbol{S}_w\boldsymbol{W})}$$

其中 $tr(.)$ 表示矩阵的迹。一个矩阵的迹是矩阵对角线的元素之和，它是一个矩阵不变量，也等于所有特征值之和。

> 还有一个常用的矩阵不变量就是矩阵的行列式，它等于矩阵的所有特征值之积。

多分类 LDA 将样本投影到 $M-1$ 维空间，因此它是一种经典的监督降维技术。之所以叫监督是因为对于每个训练样本，我们知道它所属的类别。

1.3 Python 实战

首先导入包

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets, linear_model,discriminant_analysis,cross_validation
```

在线性回归问题中，使用的数据集是scikit-learn自带的一个糖尿病病人的数据集。该数据集从糖尿病病人采样并整理后，特点如下：

- 数据集有 442 个样本；
- 每个样本有 10 个特征；
- 每个特征都是浮点数，数据都在 −0.2 ~ 0.2 之间；
- 样本的目标在整数 25 ~ 346 之间。

这里给出加载数据集的函数：

```
def load_data():
    diabetes = datasets.load_diabetes()
    return cross_validation.train_test_split(datasets.data,diabetes.target,
        test_size=0.25,random_state=0)
```

- 返回值：一个元组，元组依次是：训练样本集、测试样本集、训练样本集对应的标签值、测试样本集对应的标签值。

load_data函数加载数据集并随机切分数据集为两个部分，其中test_size指定了测试集为原始数据集的大小（比例）。

1.3.1 线性回归模型

LinearRegression是scikit-learn提供的线性回归模型，它的原型为：

```
class sklearn.linear_model.LinearRegression(fit_intercept=True, normalize=False,
    copy_X=True, n_jobs=1)
```

- 参数
  - fit_intercept：一个布尔值，指定是否需要计算 b 值。如果为False，那么不计算 b 值。

    > 当 $\vec{\boldsymbol{w}} = (w^{(1)}, w^{(2)}, \cdots, w^{(n)}, b)^T = (\vec{\boldsymbol{w}}^T, b)^T, \vec{\boldsymbol{x}} = (x^{(1)}, x^{(2)}, \cdots, x^{(n)}, 1)^T = (\vec{\boldsymbol{x}}^T, 1)^T$ 时，可以设置 fit_intercept=False。

  - normalize：一个布尔值。如果为True，那么训练样本会在回归之前会被归一化。
  - copy_X：一个布尔值。如果为True，则会复制X。
  - n_jobs：一个正数。任务并行时指定的 CPU 数量。如果为 -1 则使用所有可用的 CPU。
- 属性
  - coef_：权重向量。
  - intercept_：b 值。
- 方法
  - fit(X,y[,sample_weight])：训练模型。
  - predict(X)：用模型进行预测，返回预测值。
  - score(X,y[,sample_weight])：返回预测性能得分。设预测集为 T_{test}，真实值为 y_i，真实值的均值为 $\bar{y}$，预测值为 $\hat{y}_i$，则：

$$score = 1 - \frac{\sum_{T_{test}} (y_i - \hat{y}_i)^2}{(y_i - \bar{y})^2}$$

    - score不超过 1，但是可能为负值（预测效果太差）。
    - score越大，预测性能越好。

使用LinearRegression的函数如下：

```
def test_LinearRegression(*data):
    X_train,X_test,y_train,y_test=data
    regr = linear_model.LinearRegression()
    regr.fit(X_train, y_train)
```

```
    print('Coefficients:%s, intercept %.2f'%(regr.coef_,regr.intercept_))
    print("Residual sum of squares: %.2f"% np.mean((regr.predict(X_test) - y_test) ** 2))
    print('Score: %.2f' % regr.score(X_test, y_test))
```

- 参数：data依次指定了训练样本集、测试样本集、训练样本集对应的标签值、测试样本集对应的标签值。

该函数简单地从训练数据集中学习，然后从测试数据集中预测。我们调用该函数：

```
X_train,X_test,y_train,y_test=load_data()
test_LinearRegression(X_train,X_test,y_train,y_test)
```

输出结果如下：

```
Coefficients:[ -43.26774487 -208.67053951  593.39797213  302.89814903 -560.27689824
  261.47657106   -8.83343952  135.93715156  703.22658427   28.34844354], intercept 153.07
Residual sum of squares: 3180.20
Score: 0.36
```

可以看到测试集中预测结果的均方误差为 3180.20，预测性能得分仅为 0.36（该值越大越好，1.0 为最好）。

1.3.2 线性回归模型的正则化

前面理论部分讲到对于多元线性回归，当 $\vec{x}^T\vec{x}$ 不是满秩矩阵时存在多个解析解，它们都能使得均方误差最小化，常见的做法是引入正则化项。所谓正则化，就是对模型的参数添加一些先验假设，控制模型空间，以达到使得模型复杂度较小的目的。岭回归和 LASSO 是目前最为流行的两种线性回归正则化方法。根据不同的正则化方式，有不同的方法：

- Ridge Regression：正则化项为 $\alpha||\vec{w}||_2^2, \alpha \geqslant 0$。
- Lasso Regression：正则化项为：$\alpha||\vec{w}||_1, \alpha \geqslant 0$。
- Elastic Net：正则化项为：$\alpha\rho||\vec{w}||_1 + \frac{\alpha(1-\rho)}{2}||\vec{w}||_2^2, \alpha \geqslant 0, 1 \geqslant \rho \geqslant 0$。

其中，正则项系数 α 的选择很关键，初始值建议一开始设置为 0，先确定一个比较好的 learning rate，然后固定该 learning rate，给 α 一个值（比如 1.0），然后根据 validation accuracy，将 α 增大或者减小 10 倍（增减 10 倍是粗调节，当你确定了 α 合适的数量级后，比如 α= 0.01, 再进一步地细调节，比如调节为 0.02，0.03，0.009 之类。）

岭回归

岭回归（Ridge Regression）是一种正则化方法，通过在损失函数中加入 L2 范数惩罚项，来控制线性模型的复杂程度，从而使得模型更稳健。Ridge类实现了岭回归模型。其原型为：

```
lass sklearn.linear_model.Ridge(alpha=1.0, fit_intercept=True, normalize=False,
    copy_X=True, max_iter=None, tol=0.001, solver='auto', random_state=None)
```

❑ 参数

❍ alpha：α 值，其值越大则正则化项的占比越大。

❍ fit_intercept：一个布尔值，指定是否需要计算 b 值。如果为False，那么不计算 b 值（模型会假设你的数据已经中心化了）。

> 当 $\vec{\boldsymbol{w}} = (w^{(1)}, w^{(2)}, \cdots, w^{(n)}, b)^T = (\vec{\boldsymbol{w}}^T, b)^T, \vec{\boldsymbol{x}} = (x^{(1)}, x^{(2)}, \cdots, x^{(n)}, 1)^T = (\vec{\boldsymbol{x}}^T, 1)^T$ 时，可以设置 fit_intercept=False。

❍ max_iter：一个整数，指定最大迭代次数。如果为None，则为默认值（不同solver的默认值不同）。

❍ normalize：一个布尔值。如果为True，那么训练样本会在回归之前会被归一化。

❍ copy_X：一个布尔值。如果为True，则会复制X。

❍ solver：一个字符串，指定求解最优化问题的算法。可以为

- ❖ auto：根据数据集自动选择算法；
- ❖ svd：使用奇异值分解来计算回归系数；
- ❖ cholesky：使用scipy.linalg.solve函数来求解；
- ❖ sparse_cg：使用scipy.sparse.linalg.cg函数来求解；
- ❖ lsqr：使用scipy.sparse.linalg.lsqr函数来求解。它运算速度最快，但是可能老版本的scipy不支持；
- ❖ sag：使用Stochastic Average Gradient descent算法，求解最优化问题。

❍ tol：一个浮点数，指定判断迭代收敛与否的阈值。

❍ random_state：一个整数或者一个RandomState实例，或者None；它在solver=sag时使用。

- ❖ 如果为整数，则它指定了随机数生成器的种子。
- ❖ 如果为RandomState实例，则指定了随机数生成器。
- ❖ 如果为None，则使用默认的随机数生成器。

❑ 属性

❍ coef_：权重向量。

❍ intercept_：b 值。

❍ n_iter_：实际迭代次数。

❑ 方法

❍ fit(X,y[,sample_weight])：训练模型。

❍ predict(X)：用模型进行预测，返回预测值。

❍ score(X,y[,sample_weight])：返回预测性能得分。设预测集为 T_{test}，真实值为 y_i，真实值的均值为 $\bar{y}$，预测值为 $\hat{y}_i$，则：

$$score = 1 - \frac{\sum_{T_{test}}(y_i - \hat{y}_i)^2}{(y_i - \bar{y})^2}$$

- score不超过 1，但是可能为负值（预测效果太差）。
- score越大，预测性能越好。

使用Ridge的函数如下：

```
def test_Ridge(*data):
    X_train,X_test,y_train,y_test=data
    regr = linear_model.Ridge()
    regr.fit(X_train, y_train)
    print('Coefficients:%s, intercept %.2f'%(regr.coef_,regr.intercept_))
    print("Residual sum of squares: %.2f"% np.mean((regr.predict(X_test) - y_test) ** 2))
    print('Score: %.2f' % regr.score(X_test, y_test))
```

- 参数：data依次指定了训练样本集、测试样本集、训练样本集对应的标签值、测试样本集对应的标签值。

该函数简单地从训练数据集中学习，然后从测试数据集中预测。这里Ridge所有的参数都采用默认值。

调用该函数：

```
X_train,X_test,y_train,y_test=load_data()
test_Ridge(X_train,X_test,y_train,y_test)
```

输出结果如下：

```
Coefficients:[  21.19927911  -60.47711393  302.87575204  179.41206395    8.90911449
  -28.8080548  -149.30722541  112.67185758  250.53760873   99.57749017], intercept 152.45
Residual sum of squares: 3192.33
Score: 0.36
```

可以看到测试集中预测结果的均方误差为 3192.33，预测性能得分仅为 0.36（该值越大越好，1.0 为最好）。

下面检验不同的 α 值对于预测性能的影响，给出测试函数：

```
def test_Ridge_alpha(*data):
    X_train,X_test,y_train,y_test=data
    alphas=[0.01,0.02,0.05,0.1,0.2,0.5,1,2,5,10,20,50,100,200,500,1000]
    scores=[]
    for i,alpha in enumerate(alphas):
        regr = linear_model.Ridge(alpha=alpha)
        regr.fit(X_train, y_train)
        scores.append(regr.score(X_test, y_test))
    ## 绘图
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    ax.plot(alphas,scores)
```

```
    ax.set_xlabel(r"$\alpha$")
    ax.set_ylabel(r"score")
    ax.set_xscale('log')
    ax.set_title("Ridge")
    plt.show()
```

- 参数：data依次指定了训练样本集、测试样本集、训练样本集对应的标签值、测试样本集对应的标签值。

该函数检验了当 α 依次取值为 0.01,0.02,0.05,0.1,0.2,0.5,1,2,5,10,20,50,100,200,500,1000 时，对预测性能的影响。为了便于观察结果，将 x 轴设置为对数坐标。

调用该函数：

```
X_train,X_test,y_train,y_test=load_data()
test_Ridge_alpha(X_train,X_test,y_train,y_test)
```

输出结果如图 1.2 所示。

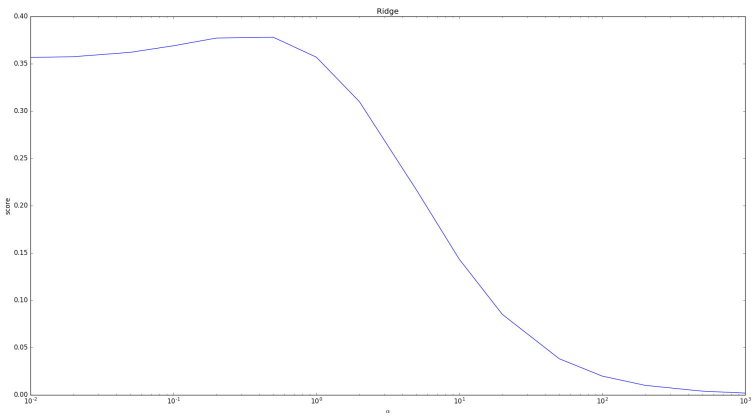


图 1.2 Ridge_alpha

可以看到，当 α 超过 1 之后，随着 α 的增长，预测性能急剧下降。这是因为 α 较大时，正则化项 $\alpha||\vec{w}||_2^2$ 影响较大，模型趋向于简单。极端情况下当 $\alpha \to \infty$ 时，$\vec{w} = \vec{0}$，从而使得正则化项 $\alpha||\vec{w}||_2^2 = 0$，此时的模型最简单，但是预测性能非常差（对所有的未知样本都预测为同一个常数 b）。

Lasso 回归

Lasso 回归和岭回归的区别在于它的惩罚项是基于 L1 范数, 因此，它可以将系数控制收缩到 0, 从而达到变量选择的效果, 这是一种非常流行的变量选择方法。Lasso类实现了Lasso回归模型，其原型为：

```
lass sklearn.linear_model.Lasso(alpha=1.0, fit_intercept=True, normalize=False,
    precompute=False, copy_X=True, max_iter=1000, tol=0.0001, warm_start=False,
```

```
positive=False, random_state=None, selection='cyclic')
```

- ❑ 参数
  - ❍ alpha：α 值。
  - ❍ fit_intercept：一个布尔值，指定是否需要计算 b 值。如果为False，那么不会计算 b 值（模型会假设你的数据已经中心化了）。

> 当 $\vec{\boldsymbol{w}} = (w^{(1)}, w^{(2)}, \cdots, w^{(n)}, b)^T = (\vec{\boldsymbol{w}}^T, b)^T, \vec{\tilde{\boldsymbol{x}}} = (x^{(1)}, x^{(2)}, \cdots, x^{(n)}, 1)^T = (\vec{\boldsymbol{x}}^T, 1)^T$ 时，可以设置 fit_intercept=False。

  - ❍ max_iter：一个整数，指定最大迭代次数。
  - ❍ normalize：一个布尔值。如果为True，那么训练样本会在回归之前会被归一化。
  - ❍ copy_X：一个布尔值。如果为True，则会复制X。
  - ❍ precompute：一个布尔值或者一个序列。它决定是否提前计算Gram矩阵来加速计算。
  - ❍ tol：一个浮点数，指定判断迭代收敛与否的阈值。
  - ❍ warm_start：一个布尔值。如果为True，那么使用前一次训练结果继续训练。否则从头开始训练。
  - ❍ positive：一个布尔值。如果为True，那么强制要求权重向量的分量都为正数。
  - ❍ selection：一个字符串，可以为'cyclic'或者'random'。它指定了当每轮迭代的时候，选择权重向量的哪个分量来更新。
    - ❖ 'random'：更新的时候，随机选择权重向量的一个分量来更新。
    - ❖ 'cyclic'：更新的时候，从前向后依次选择权重向量的一个分量来更新。
  - ❍ random_state：一个整数或者一个RandomState实例，或者None。
    - ❖ 如果为整数，则它指定了随机数生成器的种子。
    - ❖ 如果为RandomState实例，则指定了随机数生成器。
    - ❖ 如果为None，则使用默认的随机数生成器。
- ❑ 属性
  - ❍ coef_：权重向量。
  - ❍ intercept_：b 值。
  - ❍ n_iter_：实际迭代次数。
- ❑ 方法
  - ❍ fit(X,y[,sample_weight])：训练模型。
  - ❍ predict(X)：用模型进行预测，返回预测值。
  - ❍ score(X,y[,sample_weight])：返回预测性能得分。设预测集为 T_{test}，真实值为 y_i，真实值的均值为 $\bar{y}$，预测值为 $\hat{y}_i$，则：

$$score = 1 - \frac{\sum_{T_{test}} (y_i - \hat{y}_i)^2}{(y_i - \bar{y})^2}$$

    - ❖ score不超过 1，但可能为负值（预测效果太差）。

- ❖ score越大，预测性能越好。

使用Lasso的函数如下：

```
def test_Lasso(*data):
    X_train,X_test,y_train,y_test=data
    regr = linear_model.Lasso()
    regr.fit(X_train, y_train)
    print('Coefficients:%s, intercept %.2f'%(regr.coef_,regr.intercept_))
    print("Residual sum of squares: %.2f"% np.mean((regr.predict(X_test) - y_test) ** 2))
    print('Score: %.2f' % regr.score(X_test, y_test))
```

- ❑ 参数：data依次指定了训练样本集、测试样本集、训练样本集对应的标签值、测试样本集对应的标签值。

该函数简单地从训练数据集中学习，然后从测试数据集中预测。这里Lasso所有的参数都采用默认值。

调用该函数：

```
X_train,X_test,y_train,y_test=load_data()
test_Lasso(X_train,X_test,y_train,y_test)
```

输出结果如下：

```
Coefficients:[   0.          -0.         442.67992538    0.           0.           0.
   -0.           0.         330.76014648    0.        ], intercept 152.52
Residual sum of squares: 3583.42
Score: 0.28
```

可以看到测试集中的预测结果的均方误差为 3583.42，预测性能得分仅为 0.28（该值越大越好，1.0 为最好）。

下面检验不同的 α 值对于预测性能的影响，给出测试函数：

```
def test_Lasso_alpha(*data):
    X_train,X_test,y_train,y_test=data
    alphas=[0.01,0.02,0.05,0.1,0.2,0.5,1,2,5,10,20,50,100,200,500,1000]
    scores=[]
    for i,alpha in enumerate(alphas):
        regr = linear_model.Lasso(alpha=alpha)
        regr.fit(X_train, y_train)
        scores.append(regr.score(X_test, y_test))
    ## 绘图
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    ax.plot(alphas,scores)
    ax.set_xlabel(r"$\alpha$")
    ax.set_ylabel(r"score")
```

```
    ax.set_xscale('log')
    ax.set_title("Lasso")
    plt.show()
```

- 参数：data依次指定了训练样本集、测试样本集、训练样本集对应的标签值、测试样本集对应的标签值。

该函数检验了当 α 依次取值为 0.01,0.02,0.05,0.1,0.2,0.5,1,2,5,10,20,50,100,200,500,1000 时，对预测性能的影响。为了便于观察结果，我们将 x 轴设置为对数坐标。

调用该函数：

```
X_train,X_test,y_train,y_test=load_data()
test_Lasso_alpha(X_train,X_test,y_train,y_test)
```

输出结果如图 1.3 所示。可以看到，当 α 超过 1 之后，随着 α 的增长，预测性能急剧下降。原因同Ridge中的分析。

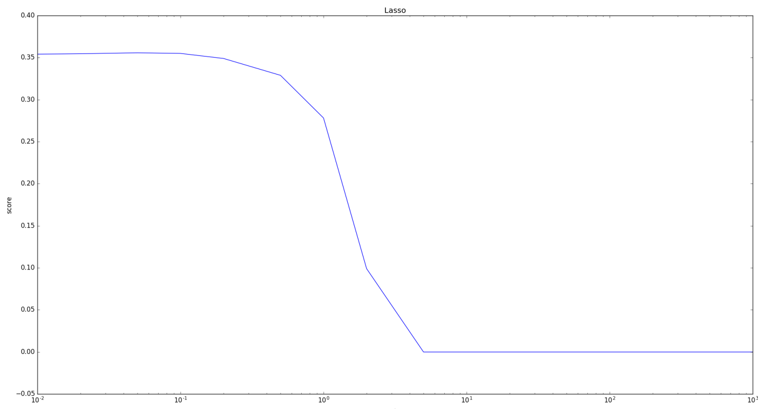


图 1.3 Lasso_alpha

ElasticNet 回归

ElasticNet 回归是对 Lasso 回归和岭回归的融合，其惩罚项是 L1 范数和 L2 范数的一个权衡。ElasticNet类实现了ElasticNet回归模型。其原型为：

```
class sklearn.linear_model.ElasticNet(alpha=1.0, l1_ratio=0.5, fit_intercept=True,
     normalize=False, precompute=False, max_iter=1000, copy_X=True, tol=0.0001,
     warm_start=False, positive=False, random_state=None, selection='cyclic')
```

- 参数
  - alpha：α 值。
  - l1_ratio：ρ 值。
  - fit_intercept：一个布尔值，指定是否需要计算 b 值。如果为False，那么不计算 b 值（模型会假设你的数据已经中心化了）。

> 当 $\vec{\boldsymbol{w}} = (w^{(1)}, w^{(2)}, \cdots, w^{(n)}, b)^T = (\vec{\boldsymbol{w}}^T, b)^T, \vec{\boldsymbol{x}} = (x^{(1)}, x^{(2)}, \cdots, x^{(n)}, 1)^T = (\vec{\boldsymbol{x}}^T, 1)^T$ 时，可以设置 fit_intercept=False。

  - max_iter：一个整数，指定最大迭代次数。
  - normalize：一个布尔值。如果为True，那么训练样本会在回归之前会被归一化。
  - copy_X：一个布尔值。如果为True，则会复制X。
  - precompute：一个布尔值或者一个序列。它决定是否提前计算Gram矩阵来加速计算。
  - tol：一个浮点数，指定判断迭代收敛与否的阈值。
  - warm_start：一个布尔值。如果为True，那么使用前一次训练结果继续训练。否则从头开始训练。
  - positive：一个布尔值。如果为True，那么强制要求权重向量的分量都为正数。
  - selection：一个字符串，可以为'cyclic'或者'random'。它指定了当每轮迭代的时候，选择权重向量的哪个分量来更新。
    - 'random'：更新的时候，随机选择权重向量的一个分量来更新。
    - 'cyclic'：更新的时候，从前向后依次选择权重向量的一个分量来更新。
  - random_state：一个整数或者一个RandomState实例，或者None。
    - 如果为整数，则它指定了随机数生成器的种子。
    - 如果为RandomState实例，则指定了随机数生成器。
    - 如果为None，则使用默认的随机数生成器。
- 属性
  - coef_：权重向量。
  - intercept_：b 值。
  - n_iter_：实际迭代次数。
- 方法
  - fit(X,y[,sample_weight])：训练模型。
  - predict(X)：用模型进行预测，返回预测值。
  - score(X,y[,sample_weight])：返回预测性能得分。设预测集为 T_{test}，真实值为 y_i，真实值的均值为 $\bar{y}$，预测值为 $\hat{y}_i$，则：

$$score = 1 - \frac{\sum_{T_{test}}(y_i - \hat{y}_i)^2}{(y_i - \bar{y})^2}$$

- ❖ score不超过 1，但可能为负值（预测效果太差）。
- ❖ score越大，预测性能越好。

使用ElasticNet的函数如下：

```
def test_ElasticNet(*data):
    X_train,X_test,y_train,y_test=data
    regr = linear_model.ElasticNet()
    regr.fit(X_train, y_train)
    print('Coefficients:%s, intercept %.2f'%(regr.coef_,regr.intercept_))
    print("Residual sum of squares: %.2f"% np.mean((regr.predict(X_test) - y_test) ** 2))
    print('Score: %.2f' % regr.score(X_test, y_test))
```

❑ 参数：data依次指定了训练样本集、测试样本集、训练样本集对应的标签值、测试样本集对应的标签值。

该函数简单地从训练数据集中学习，然后从测试数据集中预测。这里ElasticNet所有的参数都采用默认值。

调用该函数：

```
X_train,X_test,y_train,y_test=load_data()
test_ElasticNet(X_train,X_test,y_train,y_test)
```

输出结果如下：

```
Coefficients:[ 0.40560736  0.          3.76542456  2.38531508  0.58677945  0.22891647
 -2.15858149  2.33867566  3.49846121  1.98299707], intercept 151.93
Residual sum of squares: 4922.36
Score: 0.01
```

可以看到测试集中预测结果的均方误差为 4922.36，预测性能得分仅为 0.01（该值越大越好，1.0 为最好）。

下面检验不同的 α, ρ 值对于预测性能的影响，给出测试函数：

```
def test_ElasticNet_alpha_rho(*data):
    X_train,X_test,y_train,y_test=data
    alphas=np.logspace(-2,2)
    rhos=np.linspace(0.01,1)
    scores=[]
    for alpha in alphas:
            for rho in rhos:
                regr = linear_model.ElasticNet(alpha=alpha,l1_ratio=rho)
                regr.fit(X_train, y_train)
                scores.append(regr.score(X_test, y_test))
    ## 绘图
    alphas, rhos = np.meshgrid(alphas, rhos)
```

```
scores=np.array(scores).reshape(alphas.shape)
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
fig=plt.figure()
ax=Axes3D(fig)
surf = ax.plot_surface(alphas, rhos, scores, rstride=1, cstride=1, cmap=cm.jet,
    linewidth=0, antialiased=False)
fig.colorbar(surf, shrink=0.5, aspect=5)
ax.set_xlabel(r"$\alpha$")
ax.set_ylabel(r"$\rho$")
ax.set_zlabel("score")
ax.set_title("ElasticNet")
plt.show()
```

- ❑ 参数：data依次指定了训练样本集、测试样本集、训练样本集对应的标签值、测试样本集对应的标签值。

该函数在三维图形中展示了不同的 α, ρ 值对于预测性能 score的影响。

调用该函数：

```
X_train,X_test,y_train,y_test=load_data()
test_ElasticNet_alpha_rho(X_train,X_test,y_train,y_test)
```

输出结果如图 1.4 所示。可以看到随着 α 的增大，预测性能下降。因为正则化项为：$\alpha\rho||\vec{\boldsymbol{w}}||_1 + \frac{\alpha(1-\rho)}{2}||\vec{\boldsymbol{w}}||_2^2$ $,\alpha \geqslant 0, 1 \geqslant \rho \geqslant 0$。而 ρ 影响的是性能下降的速度，因为这个参数控制着 $\vec{\boldsymbol{w}}||_1$□$\vec{\boldsymbol{w}}||_2^2$ 之间的比例。

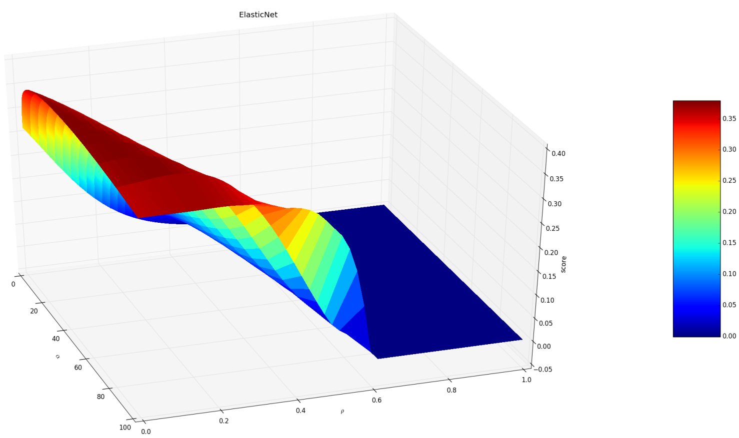


图 1.4 ElasticNet_alpha_rho

1.3.3 逻辑回归

在scikit-learn中，LogisticRegression实现了逻辑回归模型，其原型为：

```
class sklearn.linear_model.LogisticRegression(penalty='l2', dual=False, tol=0.0001,
     C=1.0, fit_intercept=True, intercept_scaling=1, class_weight=None,
    random_state=None, solver='liblinear', max_iter=100, multi_class='ovr',
    verbose=0, warm_start=False, n_jobs=1)
```

- 参数
  - penalty：一个字符串，指定了正则化策略。
    - 如果为'l2'，则优化目标函数为：$\frac{1}{2}||\vec{\boldsymbol{w}}||_2^2+CL(\vec{\boldsymbol{w}}),C>0,L(\vec{\boldsymbol{w}})$ 为极大似然函数。
    - 如果为'l1'，则优化目标函数为：$||\vec{\boldsymbol{w}}||_1+CL(\vec{\boldsymbol{w}}),C>0,L(\vec{\boldsymbol{w}})$ 为极大似然函数。
  - dual：一个布尔值。如果为True，则求解对偶形式（只在penalty='l2'且 solver='liblinear' 有对偶形式）；如果为False，则求解原始形式。
  - C：一个浮点数。它指定了罚项系数的倒数。如果它的值越小，则正则化项越大。
  - fit_intercept：一个布尔值，指定是否需要计算 b 值。如果为False，那么不会计算 b 值（模型会假设你的数据已经中心化了）。

> 当 $\vec{\boldsymbol{w}}=(w^{(1)},w^{(2)},\cdots,w^{(n)},b)^T=(\vec{\boldsymbol{w}}^T,b)^T,\vec{\boldsymbol{x}}=(x^{(1)},x^{(2)},\cdots,x^{(n)},1)^T=(\vec{\boldsymbol{x}}^T,1)^T$ 时，可以设置 fit_intercept=False。

  - intercept_scaling：一个浮点数。只有当solver='liblinear'才有意义。当采用 fit_intercept 时，相当于人造一个特征出来，该特征恒为 1，其权重为 b。在计算正则化项的时候，该人造特征也被考虑了。因此为了降低这个人造特征的影响，需要提供intercept_scaling。
  - class_weight：一个字典或者字符串'balanced'。
    - 如果为字典：则字典给出了每个分类的权重，如{class_label: weight}。
    - 如果为字符串'balanced'：则每个分类的权重与该分类在样本集中出现的频率成反比。
    - 如果未指定，则每个分类的权重都为 1。
  - max_iter：一个整数，指定最大迭代次数。
  - random_state：一个整数或者一个RandomState实例，或者None。
    - 如果为整数，则它指定了随机数生成器的种子。
    - 如果为RandomState实例，则指定了随机数生成器。
    - 如果为None，则使用默认的随机数生成器。
  - solver：一个字符串，指定了求解最优化问题的算法，可以为如下的值。
    - 'newton-cg'：使用牛顿法。
    - 'lbfgs'：使用L-BFGS拟牛顿法。
    - 'liblinear'：使用liblinear。

- ❖ 'sag'：使用Stochastic Average Gradient descent算法。

注意：

- ❖ 对于规模小的数据集，'liblinear'比较适用；对于规模大的数据集，'sag'比较适用。
- ❖ 'newton-cg'、'lbfgs'、'sag'只处理penalty='l2'的情况。

- ❍ tol：一个浮点数，指定判断迭代收敛与否的阈值。
- ❍ multi_class：一个字符串，指定对于多分类问题的策略，可以为如下的值。
  - ❖ 'ovr': 采用one-vs-rest策略。
  - ❖ 'multinomial'：直接采用多分类逻辑回归策略。
- ❍ verbose：一个正数。用于开启/关闭迭代中间输出日志功能。
- ❍ warm_start：一个布尔值。如果为True，那么使用前一次训练结果继续训练。否则从头开始训练。
- ❍ n_jobs：一个正数。指定任务并行时的 CPU 数量。如果为 −1 则使用所有可用的 CPU。

- ❑ 属性
  - ❍ coef_：权重向量。
  - ❍ intercept_：b 值。
  - ❍ n_iter_：实际迭代次数。
- ❑ 方法
  - ❍ fit(X,y[,sample_weight])：训练模型。
  - ❍ predict(X)：用模型进行预测，返回预测值。
  - ❍ predict_log_proba(X)：返回一个数组，数组的元素依次是X预测为各个类别的概率的对数值。
  - ❍ predict_proba(X)：返回一个数组，数组的元素依次是X预测为各个类别的概率值。
  - ❍ score(X,y[,sample_weight])：返回在 (X,y)上预测的准确率 (accuracy)。

为了使用逻辑回归模型，我们对鸢尾花进行分类。鸢尾花数据集一共有 150 个数据，这些数据分为 3 类（分别为setosa，versicolor，virginica），每类 50 个数据。每个数据包含 4 个属性：萼片 (sepal) 长度、萼片宽度、花瓣 (petal) 长度、花瓣宽度。

首先加载数据，将 load_data()函数修改为：

```
def load_data():
    iris=datasets.load_iris()
    X_train=iris.data
    y_train=iris.target
    return cross_validation.train_test_split(X_train, y_train,test_size=0.25,
        random_state=0,stratify=y_train)
```

在这里采用分层采样。因为原始数据集中，前 50 个样本都是类别 0，中间 50 个样本都是类别 1，最后 50 个类别都是类别 2。如果不采取分层采用，那么最后切分得到的测试数据集就不是无偏的了。使用LogisticRegression的函数如下：

```
def test_LogisticRegression(*data):
    X_train,X_test,y_train,y_test=data
    regr = linear_model.LogisticRegression()
    regr.fit(X_train, y_train)
    print('Coefficients:%s, intercept %s'%(regr.coef_,regr.intercept_))
    print('Score: %.2f' % regr.score(X_test, y_test))
```

- 参数：data依次指定了训练样本集、测试样本集、训练样本集对应的标签值、测试样本集对应的标签值。

该函数简单地从训练数据集中学习，然后从测试数据集中预测。这里LogisticRegression所有的参数都采用默认值。

我们调用该函数：

```
X_train,X_test,y_train,y_test=load_data()
test_LogisticRegression(X_train,X_test,y_train,y_test)
```

输出结果如下：

```
Coefficients:[[ 0.40769719  1.32793253 -2.12687162 -0.96614355]
 [ 0.1932691  -1.31070419  0.60821724 -1.19814744]
 [-1.50100362 -1.33529511  2.16377642  2.23963779]],
intercept [ 0.24462118  1.13229922 -1.08042606]
Score: 0.97
```

可以看到测试集中的预测结果性能得分为 0.97（即预测准确率为 97%）。

下面考察multi_class参数对分类结果的影响。默认采用的是one-vs-rest策略，但是逻辑回归模型原生就支持多类分类，给出的测试函数为：

```
def test_LogisticRegression_multinomial(*data):
    X_train,X_test,y_train,y_test=data
    regr = linear_model.LogisticRegression(multi_class='multinomial',solver='lbfgs')
    regr.fit(X_train, y_train)
    print('Coefficients:%s, intercept %s'%(regr.coef_,regr.intercept_))
    print('Score: %.2f' % regr.score(X_test, y_test))
```

注意：只有solver为牛顿法或者拟牛顿法才能配合multi_class='multinomial'，否则报错。

调用该函数：

```
X_train,X_test,y_train,y_test=load_data()
test_LogisticRegression_multinomial(X_train,X_test,y_train,y_test)
```

测试结果如下：

```
Coefficients:[[-0.36834533  0.84161813 -2.27865338 -0.98934494]
 [ 0.34136192 -0.33359843 -0.031646   -0.8294743 ]
 [ 0.0269834  -0.5080197   2.31029938  1.81881924]],
intercept [  8.77142226   2.34153563 -11.11295788]
Score: 1.00
```

可以看到在这个问题中，多分类策略进一步提升了预测准确率。这里准确率提升到100%，说明对于测试集的数据，LogisticRegression分类器完全预测正确。

最后，考察参数C对分类模型的预测性能的影响。C是正则化项系数的倒数，它越小则正则化项的权重越大。给出的测试函数为：

```
def test_LogisticRegression_C(*data):
    X_train,X_test,y_train,y_test=data
    Cs=np.logspace(-2,4,num=100)
    scores=[]
    for C in Cs:
        regr = linear_model.LogisticRegression(C=C)
        regr.fit(X_train, y_train)
        scores.append(regr.score(X_test, y_test))
    ## 绘图
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    ax.plot(Cs,scores)
    ax.set_xlabel(r"C")
    ax.set_ylabel(r"score")
    ax.set_xscale('log')
    ax.set_title("LogisticRegression")
    plt.show()
```

测试结果如图 1.5 所示。可以看到随着C的增大（即正则化项减小），LogisticRegression的预测准确率上升。当 C增大到一定程度（即正则化项减小到一定程度），LogisticRegression的预测准确率维持在较高的水准保持不变。

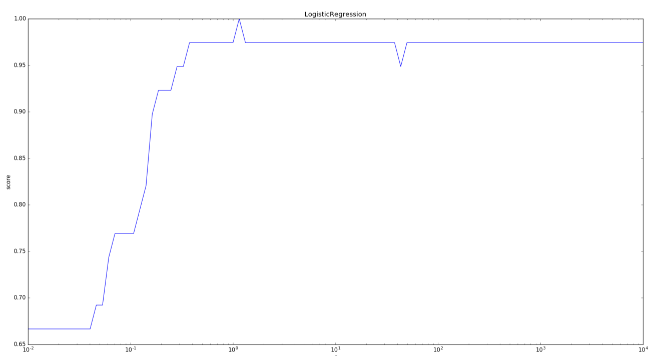


图 1.5　LogisticRegression_C

1.3.4 线性判别分析

在scikit-learn中，LinearDiscriminantAnalysis实现了线性判别分析模型。其原型为：

```
class sklearn.discriminant_analysis.LinearDiscriminantAnalysis(solver='svd',
shrinkage=None, priors=None, n_components=None, store_covariance=False, tol=0.0001)
```

- ❑ 参数
  - ❍ solver：一个字符串，指定求解最优化问题的算法。可以为
    - ❖ 'svd'：奇异值分解。对于有大规模特征的数据，推荐用这种算法。
    - ❖ 'lsqr'：最小平方差算法，可以结合shrinkage参数。
    - ❖ 'eigen'：特征值分解算法，可以结合shrinkage参数。
  - ❍ shrinkage：字符串'auto'或者浮点数或者None。该参数通常在训练样本数量小于特征数量的场合下使用。该参数只有在solver=lsqr或者eigen下才有意义。
    - ❖ 字符串'auto'：根据Ledoit-Wolf引理来自动决定 shrinkage参数的大小。
    - ❖ None：不使用shrinkage参数。
    - ❖ 浮点数（位于 0~1 之间）：指定shrinkage参数。
  - ❍ priors：一个数组，数组中的元素依次指定了每个类别的先验概率。如果为None，则认为每个类的先验概率都是等可能的。
  - ❍ n_components：一个整数。指定了数据降维后的维度（该值必须小于 n_classes-1）。
  - ❍ store_covariance：一个布尔值。如果为True，则需要额外计算每个类别的协方差矩阵 Σ_i。
  - ❍ tol：一个浮点值。它指定了用于SVD算法中评判迭代收敛的阈值。
- ❑ 属性
  - ❍ coef_：权重向量。
  - ❍ intercept_：b 值。
  - ❍ covariance_：一个数组，依次给出了每个类别的协方差矩阵。
  - ❍ means_：一个数组，依次给出了每个类别的均值向量。
  - ❍ xbar_：给出了整体样本的均值向量。
  - ❍ n_iter_：实际迭代次数。
- ❑ 方法
  - ❍ fit(X,y)：训练模型。
  - ❍ predict(X)：用模型进行预测，返回预测值。
  - ❍ predict_log_proba(X)：返回一个数组，数组的元素依次是X预测为各个类别的概率的对数值。
  - ❍ predict_proba(X)：返回一个数组，数组的元素依次是X预测为各个类别的概率值。
  - ❍ score(X,y[,sample_weight])：返回在 (X,y)上预测的准确率 (accuracy)。

这里还是使用鸢尾花数据集，先给出使用LinearDiscriminantAnalysis的函数如下：

```
def test_LinearDiscriminantAnalysis(*data):
    X_train,X_test,y_train,y_test=data
    lda = discriminant_analysis.LinearDiscriminantAnalysis()
    lda.fit(X_train, y_train)
    print('Coefficients:%s, intercept %s'%(lda.coef_,lda.intercept_))
    print('Score: %.2f' % lda.score(X_test, y_test))
```

调用该函数：

```
X_train,X_test,y_train,y_test=load_data()
test_LinearDiscriminantAnalysis(X_train,X_test,y_train,y_test)
```

测试结果如下：

```
Coefficients:[[  7.08232947   9.34752865 -14.9939558  -20.80332605]
 [ -2.17411651  -3.40669222   4.461538     2.72022083]
 [ -4.90821296  -5.94083643  10.5324178   18.08310522]],
intercept [-15.83172789   0.23565814 -31.01542519]
Score: 1.00
```

对测试集的预测准确率为 1.0，说明LinearDiscriminantAnalysis对于测试集的预测完全正确。

现在来检查一下原始数据集在经过线性判别分析LDA之后的数据集的情况。给出绘制LDA降维之后的数据集的函数：

```
def plot_LDA(converted_X,y):
    from mpl_toolkits.mplot3d import Axes3D
    fig=plt.figure()
    ax=Axes3D(fig)
    colors='rgb'
    markers='o*s'
    for target,color,marker in zip([0,1,2],colors,markers):
        pos=(y==target).ravel()
        X=converted_X[pos,:]
        ax.scatter(X[:,0], X[:,1], X[:,2],color=color,marker=marker,
            label="Label %d"%target)
    ax.legend(loc="best")
    fig.suptitle("Iris After LDA")
    plt.show()
```

然后调用该函数：

```
X_train,X_test,y_train,y_test=load_data()
X=np.vstack((X_train,X_test))
Y=np.vstack((y_train.reshape(y_train.size,1),y_test.reshape(y_test.size,1)))
```

```
lda = discriminant_analysis.LinearDiscriminantAnalysis()
lda.fit(X, Y)
converted_X=np.dot(X,np.transpose(lda.coef_))+lda.intercept_
plot_LDA(converted_X,Y)
```

执行结果如图 1.6 所示。可以看到经过线性判别分析之后，不同种类的鸢尾花之间的间隔较远；相同种类的鸢尾花之间已经相互聚集。

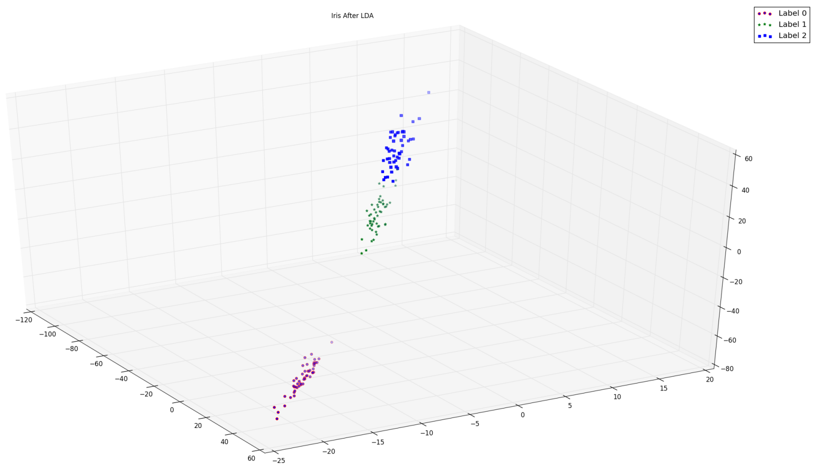


图 1.6 iris_after_LDA

接下来考察不同的solver对预测性能的影响。给出测试函数为：

```
def test_LinearDiscriminantAnalysis_solver(*data):
    X_train,X_test,y_train,y_test=data
    solvers=['svd','lsqr','eigen']
    for solver in solvers:
        if(solver=='svd'):
            lda = discriminant_analysis.LinearDiscriminantAnalysis(solver=solver)
        else:
            lda = discriminant_analysis.LinearDiscriminantAnalysis(solver=solver,
            shrinkage=None)
        lda.fit(X_train, y_train)
        print('Score at solver=%s: %.2f' %(solver, lda.score(X_test, y_test)))
```

调用该函数：

```
X_train,X_test,y_train,y_test=load_data()
test_LinearDiscriminantAnalysis_solver(X_train,X_test,y_train,y_test)
```

测试结果如下。可以看到三者没有差别。

```
Score at solver=svd: 1.00
Score at solver=lsqr: 1.00
Score at solver=eigen: 1.00
```

最后考察在solver=lsqr中引入抖动。引入抖动相当于引入了正则化项。给出测试函数为：

```
def test_LinearDiscriminantAnalysis_shrinkage(*data):
    X_train,X_test,y_train,y_test=data
    shrinkages=np.linspace(0.0,1.0,num=20)
    scores=[]
    for shrinkage in shrinkages:
        lda = discriminant_analysis.LinearDiscriminantAnalysis(solver='lsqr',
            shrinkage=shrinkage)
        lda.fit(X_train, y_train)
        scores.append(lda.score(X_test, y_test))
    ## 绘图
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    ax.plot(shrinkages,scores)
    ax.set_xlabel(r"shrinkage")
    ax.set_ylabel(r"score")
    ax.set_ylim(0,1.05)
    ax.set_title("LinearDiscriminantAnalysis")
    plt.show()
```

调用该函数：

```
X_train,X_test,y_train,y_test=load_data()
test_LinearDiscriminantAnalysis_shrinkage(X_train,X_test,y_train,y_test)
```

测试结果如图 1.7 所示。可以发现随着shrinkage的增大，模型的预测准确率会随之下降。

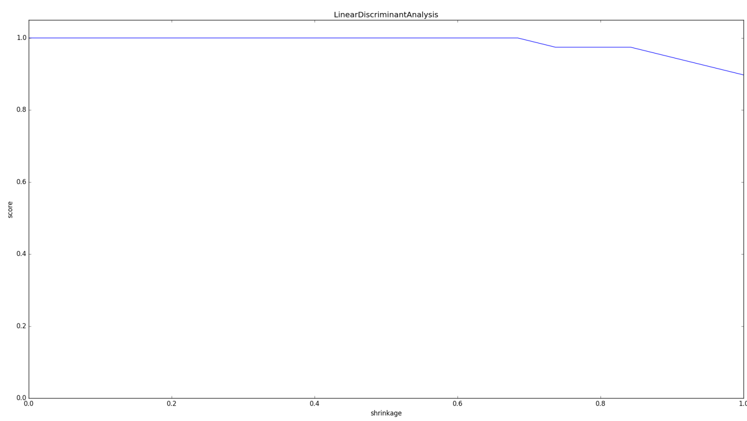


图 1.7 LinearDiscriminantAnalysis_shrinkage

第 2 章

决策树

2.1 概述

决策树decision tree是功能强大而且相当受欢迎的分类和预测方法，它是一种有监督的学习算法，以树状图为基础，其输出结果为一系列简单实用的规则，故得名决策树。决策树就是一系列的 if-then 语句，决策树可以用于分类问题，也可以用于回归问题。在讲解原理的时候为了表述方便，我们以分类问题为例。

决策树模型基于特征对实例进行分类，它是一种树状结构。决策树的优点是可读性强，分类速度快。学习决策树时，通常采用损失函数最小化原则。

在本章中，训练集用 D 表示，T 表示一棵决策树。

2.2 算法笔记精华

2.2.1 决策树原理

决策树是一个贪心算法，即在特性空间上执行递归的二元分割，决策树由节点和有向边组成。内部节点表示一个特征或者属性；叶子节点表示一个分类。使用决策树进行分类时，将实例分配到叶节点的类中，该叶节点所属的类就是该节点的分类。

决策树可以表示给定特征条件下，类别的条件概率分布。将特征空间划分为互不相交的单元 $S_1, S_2, \cdots, S_m$。设某个单元 S_i 内部有 N_i 个样本点，则它定义了一个条件概率分布 $P(y = c_k/X), X \in S_i$，$c_k, k = 1, 2, \cdots, K$ 为第 k 个分类。

- 每个单元对应于决策树的一条路径。
- 所有单元的条件概率分布构成了决策树所代表的条件概率分布。

- ❑ 在单元 S_i 内部有 N_i 个样本点，但是整个单元都属于类 $\hat{c}_k$。其中，$\hat{c}_k = \arg_{c_k}\max P(y = c_k/X), X \in S_i$。即单元 S_i 内部的 N_i 个样本点，哪个分类占优，则整个单元都属于该类。

2.2.2 构建决策树的 3 个步骤

构建决策树通常包括 3 个步骤：

- ❑ 特征选择；
- ❑ 决策树生成；
- ❑ 决策树剪枝。

假设给定训练集 $D = \{(\vec{x}_1, y_1), (\vec{x}_2, y_2), \cdots, (\vec{x}_N, y_N)\}$，其中 $\vec{x}_i = (x_i^{(1)}, x_i^{(2)}, \cdots, x_i^{(n)})$ 为输入实例，n 为特征个数；$y_i \in \{1, 2, \cdots, K\}$ 为类标记，$i = 1, 2, \cdots, N; N$ 为样本容量。构建决策树的目标是，根据给定的训练数据集学习一个决策树模型。

构建决策树时通常是将正则化的极大似然函数作为损失函数，其学习目标是损失函数为目标函数的最小化。构建决策树的算法通常是递归地选择最优特征，并根据该特征对训练数据进行分割，其步骤如下。

- ❑ 构建根节点，所有训练样本都位于根节点。
- ❑ 选择一个最优特征。通过该特征将训练数据分割成子集，确保各个子集有最好的分类，但要考虑下列两种情况：
  - ❍ 若子集已能够被“较好地”分类，则构建叶节点，并将该子集划分到对应的叶节点去；
  - ❍ 若某个子集不能够被“较好地”分类，则对该子集继续划分。
- ❑ 递归直至所有训练样本都被较好地分类，或者没有合适的特征为止。是否“较好地”分类，可通过后面介绍的指标来判断。

通过该步骤生成的决策树对训练样本有很好的分类能力，但是我们需要的是对未知样本的分类能力。因此通常需要对已生成的决策树进行剪枝，从而使得决策树具有更好的泛化能力。剪枝过程是去掉过于细分的叶节点，从而提高泛化能力。

特征选择

特征选择就是选取有较强分类能力的特征。分类能力通过信息增益或者信息增益比来刻画。选择特征的标准是找出局部最优的特征作为判断进行切分，取决于切分后节点数据集合中类别的有序程度（纯度），划分后的分区数据越纯，切分规则越合适。衡量节点数据集合的纯度有：熵、基尼系数和方差。熵和基尼系数是针对分类的，方差是针对回归的。

先给出熵entropy的定义。设 X 是一个离散型随机变量，其概率分布为：

$$P(X = \vec{x}_i) = p_i, i = 1, 2, \cdots, n$$

则随机变量 X 的熵为：

$$H(X) = -\sum_{i=1}^{n} p_i \log p_i$$

其中，定义 $0 \log 0 = 0$。

当随机变量 X 只取两个值时，X 的分布为：

$$P(X = 1) = p$$
$$P(X = 0) = 1 - p, 0 \leqslant p \leqslant 1$$

此时熵为：$H(P) = -p \log p - (1 - p) \log(1 - p), 0 \leqslant p \leqslant 1$

- ❑ 当 $p = 0$ 或者 $p = 1$ 时，熵最小（为 0），此时随机变量不确定性最小。
- ❑ 当 $p = 0.5$ 时，熵最大（为 1），此时随机变量不确定性最大。

熵 entropy $H(P)$ 的函数图像如图 2.1 所示。

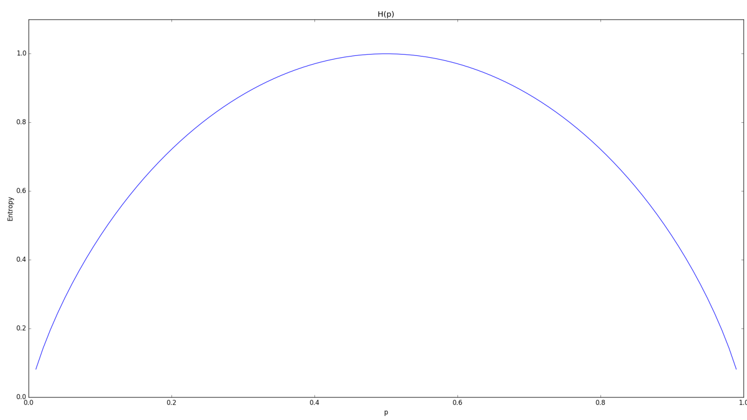


图 2.1 entropy

设随机变量 (X, Y)，其联合概率分布为：$P(X = \vec{x}_i, Y = y_j) = p_{ij}, i = 1, 2, \cdots, n; j = 1, 2, \cdots, m$。则条件熵 $H(Y/X)$ 定义为：

$$H(Y/X) = \sum_{i=1}^{n} P_X(X = \vec{x}_i) H(Y/X = \vec{x}_i)$$

其中，$P_X(X = \vec{x}_i) = \sum_Y P(X = \vec{x}_i, Y)$

- 当熵中的概率由数据估计得到时，称之为经验熵。
- 当条件熵中的概率由数据估计得到时，称之为经验条件熵。

> 其算法见本书后面的描述。

对于数据集 D，我们通过 $H(Y)$ 来刻画数据集 D 的不确定程度。当数据集 D 中的所有样本都是同一种类别时，$H(Y)=0$ 。也将 $H(Y)$ 记作 $H(D)$ 。给定特征 A 和训练数据集 D，定义信息增益 $g(D,A)$ 为：$g(D,A)=H(D)-H(D/A)$ 。

信息增益刻画的是由于特征 A 而使得对数据集 D 的分类的不确定性减少的程度。构建决策树选择信息增益大的特征来划分数据集。

这里给出计算信息增益的算法。假设训练数据集为 D，N 为其训练数据集容量。假设有 K 个类别依次为 $c_k, k=1,2,\cdots,K$。设 $|C_k|$ 为属于类 c_k 的样本个数。

设特征 A 是离散的，且有 n 个不同的取值：$\{a_1,a_2,\cdots,a_n\}$ ，根据特征 A 的取值将 D 划分出 n 个子集：$D_1,D_2,\cdots,D_n$，N_i 为对应的 D_i 中的样本个数。

设集合 D_i 中属于类 c_k 的样本集合为：D_{ik} ，其容量为：N_{ik} ，信息增益算法如下。

- 输入
  - 训练数据集 D。
  - 特征 A。
- 输出：信息增益 $g(D,A)$。
- 算法步骤
  - 根据下式计算数据集 D 的经验熵 $H(D)$ 。它就是训练数据集 D 中，分类 Y 的概率估计 $\hat{P}(Y=c_k)=\frac{|C_k|}{N}$ 计算得到的经验熵。

$$H(D)=-\sum_{k=1}^{K}\frac{|C_k|}{N}\log\frac{|C_k|}{N}$$

  - 根据下式计算特征 A 对于数据集 D 的经验条件熵 $H(D/A)$。它使用了特征 A 的概率估计：$\hat{P}(X^{(A)}=a_i)=\frac{N_i}{N}$，以及经验条件熵：$\hat{H}(D/X^{(A)}=a_i)=\sum_{k=1}^{K}-(\frac{N_{ik}}{N_i}\log\frac{N_{ik}}{N_i})$（其中使用了条件概率估计 $\hat{P}(Y=c_k/X^{(A)}=a_i)=\frac{N_{ik}}{N_i}$，意义是：在子集 D_i 中，Y 的分布）

$$H(D/A)=\sum_{i=1}^{n}\frac{N_i}{N}\sum_{k=1}^{K}-(\frac{N_{ik}}{N_i}\log\frac{N_{ik}}{N_i})$$

  - 根据下式计算信息增益：

$$g(D,A)=H(D)-H(D/A)$$

熵越大，则表示越混乱；熵越小，表示越有序。因此信息增益表示混乱的减少程度（或者说是有序的增加程度）。

以信息增益作为划分训练集的特征选取方案，存在偏向于选取值较多的特征的问题。公式：

$$g(D,A)=H(D)-H(D/A)=H(D)-\sum_{i=1}^{n}\frac{N_i}{N}\sum_{k=1}^{K}-(\frac{N_{ik}}{N_i}\log\frac{N_{ik}}{N_i})$$

在极限情况下，特征 A 将每一个样本一一对应到对应的节点当中去的时候（即每个节点中有且仅有一个样本），此时 $\frac{N_{ik}}{N_i}=1, i=1,2,\cdots,n$，条件熵部分为 0。而条件熵的最小值为 0，这意味着该情况下的信息增益达到了最大值。然而，我们知道这个特征 A 显然不是最佳的选择。

可以通过定义信息增益比来解决。特征 A 对训练集 D 的信息增益比 $g_R(D,A)$ 定义为：

$$g_R(D,A)=\frac{g(D,A)}{H_A(D)}$$

$$H_A(D)=-\sum_{i=1}^{n}\frac{N_i}{N}\log\frac{N_i}{N}$$

$H_A(D)$ 刻画了特征 A 对训练集 D 的分辨能力。但是这不表征它对类别的分辨能力。比如 A 将 D 切分成了 2 块 D_1 和 D_2，那么很有可能 $H(D)=H(D_1)=H(D_2)$（如每个子集 D_i 中各类别样本的比例与 D 中各类别样本的比例相同）。

决策树生成

基本的决策树的生成算法中，典型的有ID3生成算法和C4.5生成算法，它们生成树的过程大致相似。ID3 是采用的信息增益作为特征选择的度量，而 C4.5 则采用信息增益比。

ID3 生成算法应用信息增益准则选择特征，其算法描述如下。

- 输入
  - 训练数据集 D。
  - 特征集 $\mathcal{A}$。
  - 特征信息增益阈值 $\varepsilon>0$。
- 输出：决策树 T。
- 算法步骤
  - 若 D 中所有实例均属于同一类 c_k，则 T 为单节点树，并将 c_k 作为该节点的类标记，返回 T。这是一种特殊情况：D 的分类集合只有一个分类。
  - 若 $\mathcal{A}=\phi$，则 T 为单节点树，将 D 中实例数最大的类 c_k 作为该节点的类标记，返回 T（即多数表决）。这也是一种特殊情况：D 的特征集合为空。

- ❍ 否则计算 $g(D, A_i)$，其中 $A_i \in \mathcal{A}$ 为特征集合中的各个特征，选择信息增益最大的特征 A_g。
- ❍ 判断 A_g 的信息增益如下。
  - ❖ 若 $g(D, A_g) < \varepsilon$ ，则置 T 为单节点树，将 D 中实例数最大的类 c_k 作为该节点的类标记，返回 T。

> 如果不设置特征信息增益的下限，则可能会使得每个叶子都只有一个样本点，从而划分得太细。

  - ❖ 若 $g(D, A_g) \geqslant \varepsilon$ ，则对 A_g 特征的每个可能取值 a_i , 根据 $A_g = a_i$ 将 D 划分为若干个非空子集 D_i ，将 D_i 中实例数最大的类作为标记，构建子节点，由子节点及其子节点构成树 T ，返回 T。
- ❍ 对第 i 个子节点，以 D_i 为训练集，以 $A - \{A_g\}$ 为特征集，递归地调用前面的步骤，得到子树 T_i，返回 T_i。

`C4.5` 生成算法应用信息增益比来选择特征，其算法描述如下。

- ❑ 输入
  - ❍ 训练数据集 D。
  - ❍ 特征集 $\mathcal{A}$。
  - ❍ 特征信息增益比的阈值 $\varepsilon > 0$。
- ❑ 输出：决策树 T。
- ❑ 算法步骤
  - ❍ 若 D 中所有实例均属于同一类 c_k，则 T 为单节点树，并将 c_k 作为该节点的类标记，返回 T。这是一种特殊情况：D 的分类集合只有一个分类。
  - ❍ 若 $\mathcal{A} = \phi$，则 T 为单节点树，将 D 中实例数最大的类 c_k 作为该节点的类标记，返回 T（即多数表决）。这也是一种特殊情况：D 的特征集合为空。
  - ❍ 否则计算 $g_R(D, A_i)$，其中 $A_i \in \mathcal{A}$ 为特征集合中的各个特征，选择信息增益比最大的特征 A_g。
  - ❍ 判断 A_g 的信息增益比如下。
    - ❖ 若 $g_R(D, A_g) < \varepsilon$ ，则置 T 为单节点树，将 D 中实例数最大的类 c_k 作为该节点的类标记（即多数表决），返回 T。
    - ❖ 若 $g_R(D, A_g) \geqslant \varepsilon$ ，则对 A_g 特征的每个可能取值 a_i ，根据 $A_g = a_i$ 将 D 划分为若干个非空子集 D_i ，将 D_i 中实例数最大的类作为标记（即多数表决），构建子节点，由子节点及其子节点构成树 T ，返回 T。
  - ❍ 对第 i 个子节点，以 D_i 为训练集，以 $A - \{A_g\}$ 为特征集，递归地调用前面的步骤，得到子树 T_i, 返回 T_i。

几点说明：

- C4.5 算法继承了 ID3 算法的优点，并在以下几方面对 ID3 算法进行了改进：
  - 用信息增益率来选择属性，克服了用信息增益选择属性时偏向选择取值多的属性的不足；
  - 在树构造过程中进行剪枝；
  - 能够完成对连续属性的离散化处理；
  - 能够对不完整数据进行处理。
- C4.5 算法有如下优点：产生的分类规则易于理解，准确率较高。其缺点是：在构造树的过程中，需要对数据集进行多次的顺序扫描和排序，因而导致算法的低效。此外，C4.5 只适合于能够驻留于内存的数据集，当训练集大得无法在内存容纳时程序无法运行。
- 决策树可能只是用到特征集中的部分特征。
- C4.5 和 ID3 两个算法只有树的生成算法，生成的树容易产生过拟合。即对训练集匹配很好，但是预测测试集效果较差。

树剪枝

决策树需要剪枝的原因是：决策树生成算法生成的树对训练数据的预测很准确，但是对于未知的数据分类却很差，这就产生过拟合的现象。其实，原理都是一样的，决策树的构建是直到没有特征可选或者信息增益很小，这就导致构建的决策树模型过于复杂，而复杂的模型是在训练数据集上建立的，所以对于测试集往往造成分类的不准确，这就是过拟合。发生过拟合是由于决策树太复杂，解决过拟合的方法是控制模型的复杂度，对于决策树来说就是简化模型，称为剪枝。

决策树剪枝过程是从已生成的决策树上裁掉一些子树或者叶节点。剪枝的目标是通过极小化决策树的整体损失函数或代价函数来实现的。

决策树剪枝的目的是通过剪枝来提高泛化能力。剪枝的思路就是在决策树对训练数据的预测误差和数据复杂度之间找到一个平衡。

设树 T 的叶节点个数为 $|T_f|$，t 为树的叶节点，该叶节点有 N_t 个样本点，其中属于 c_k 类的样本点有 N_{tk}，　$k=1,2,\cdots,K$ 个。则有：$\sum_{k=1}^{K} N_{tk} = N_t$。

令 $H(t)$ 为叶节点 t 上的经验熵，$\alpha \geqslant 0$ 为参数，则决策树 T 的损失函数定义为：

$$C_\alpha(T) = \sum_{t=1}^{|T_f|} N_t H(t) + \alpha|T_f|H(t) = -\sum_{k=1}^{K} \frac{N_{tk}}{N_t} \log \frac{N_{tk}}{N_t}$$

令:

$$C(T) = \sum_{t=1}^{|T_f|} N_t H(t) = -\sum_{t=1}^{|T_f|} \sum_{k=1}^{K} N_{tk} \log \frac{N_{tk}}{N_t}$$

则：$C_\alpha(T)=C(T)+\alpha|T_f|$，其中 $\alpha|T_f|$ 为正则化项，$C(T)$ 表示预测误差。

- $C(T)=0$ 意味着 $N_{tk}=N_t$，即每个节点 t 内的样本都是纯的（即单一的分类，而不是杂的）。
- 决策树划分得越细致，则 T 的叶子节点越多，$|T_f|$ 越大；$|T_f|$ 小于等于样本集的数量，当取等号时，树 T 的每个叶子节点只有一个样本点。
- 参数 α 控制预测误差与模型复杂度之间的关系：
  - 较大的 α 会选择较简单的模型；
  - 较小的 α 会选择较复杂的模型；
  - $\alpha=0$ 只考虑训练数据与模型的拟合程度，不考虑模型复杂度。

剪枝算法的描述如下。

- 输入
  - 生成树 T。
  - 参数 α。
- 输出：剪枝树 T_α。
- 算法步骤如下。
  - 计算每个节点的经验熵；
  - 递归地从树的叶节点向上回退：
    设一组叶节点回退到父节点之前与之后的整棵树分别为 T_t 与 T_t'，对应的损失函数值分别为 $C_\alpha(T_t)$ 与 $C_\alpha(T_t')$。若 $C_\alpha(T_t)'\leqslant C_\alpha(T_t)$，则进行剪枝并将父节点变成新的叶节点。
- 递归进行上一步，直到不能继续为止，得到损失函数最小的子树 T_α。

2.2.3 CART 算法

分类与回归树(Classfification And Regression Tree，CART)模型也是一种决策树模型，它既可用于分类，也可用于回归。其学习算法分为如下两步。

- 决策树生成：用训练数据生成决策树，生成树尽可能地大；
- 决策树剪枝：基于损失函数最小化的标准，用验证数据对生成的决策树剪枝。

分类与回归树模型采用不同的最优化策略。CART回归生成树用平方误差最小化策略，CART分类生成树采用基尼指数最小化策略。

CART 回归树

给定训练数据集 $D=\{(\vec{x}_1,y_1),(\vec{x}_2,y_2),\cdots,(\vec{x}_N,y_N)\},y_i\in\mathbb{R}$。设已经将输入空间划分为 M 个单元 $R_1,R_2,\cdots,R_M$，且在单元 R_m 上输出值为 $c_m,m=1,2,\cdots,M$。则回归树模型为：

$$f(\vec{\boldsymbol{x}})=\sum_{m=1}^{M} c_m I(\vec{\boldsymbol{x}} \in R_m)$$

其中，$I(\cdot)$ 为示性函数。

如果给定输入空间的一个划分，回归树在训练数据集上的误差（平方误差）为：

$$\sum_{\vec{\boldsymbol{x}}_i \in R_m} (y_i - f(\vec{\boldsymbol{x}}_i))^2$$

基于平方误差最小的准则，可以求解出每个单元上的最优输出值 $\hat{c}_m$ ：$\hat{c}_m = \text{ave}(y_i \mid \vec{\boldsymbol{x}}_i \in R_m)$ 。它就是 R_m 上所有输入样本对应的输出 y_i 的平均值。

现在需要找到最佳的划分，使得该划分对应的回归树的平方误差在所有划分中最小。设 $\vec{\boldsymbol{x}}_i=(x_i^{(1)}, x_i^{(2)}, \cdots, x_i^{(k)})$，即输入为 k 维。选择第 j 维 $x_i^{(j)}$，它的取值 s 作为切分变量和切分点。定义两个区域：

$$R_1(j,s)=\{\vec{\boldsymbol{x}} \mid x^{(j)} \leqslant s\}$$
$$R_2(j,s)=\{\vec{\boldsymbol{x}} \mid x^{(j)} > s\}$$

然后寻求最优切分变量 j 和最优切分点 s。即求解：

$$\min_{j,s}[\min_{c_1} \sum_{\boldsymbol{x}_i \in R_1(j,s)} (y_i - c_1)^2 + \min_{c_2} \sum_{\boldsymbol{x}_i \in R_2(j,s)} (y_i - c_2)^2]$$

对于给定的维度 j 可以找到最优切分点 s 。同时：

$$\hat{c}_1 = \text{ave}(y_i \mid \vec{\boldsymbol{x}}_i \in R_1(j,s))\hat{c}_2 = \text{ave}(y_i \mid \vec{\boldsymbol{x}}_i \in R_2(j,s))$$

问题是如何求解 j ? 首先遍历所有的维度，找到最优切分维度 j ；然后对该维度找到最优切分点 s 构成一个 (j,s) 对，并将输入空间划分为两个区域。然后在子区域中重复划分过程，直到满足停止条件为止。这样的回归树称为最小二乘回归树。

最小二乘回归树生成算法的描述如下。

- 输入
  - 训练数据集 D。
  - 停止计算条件。
- 输出：CART 回归树 $f(\vec{\boldsymbol{x}})$。
- 算法步骤
  - 选择数据集 D 的最优切分维度 j 和切分点 s。即求解：

$$\min_{j,s}[\min_{c_1}\sum_{\vec{\boldsymbol{x}}_i\in R_1(j,s)}(y_i-c_1)^2+\min_{c_2}\sum_{\vec{\boldsymbol{x}}_i\in R_2(j,s)}(y_i-c_2)^2]$$

❖ 求解方法：遍历 j,s 找到使上式最小的 (j,s) 对。

❍ 用选定的 (j,s) 划分区域并决定相应的输出值：

$$R_1(j,s)=\{\vec{\boldsymbol{x}}\mid x^{(j)}\leqslant s\}$$
$$R_2(j,s)=\{\vec{\boldsymbol{x}}\mid x^{(j)}>s\}$$
$$\hat{c}_1=\mathrm{ave}(y_i\mid\vec{\boldsymbol{x}}_i\in R_1(j,s))$$
$$\hat{c}_2=\mathrm{ave}(y_i\mid\vec{\boldsymbol{x}}_i\in R_2(j,s))$$

❍ 对子区域 R_1,R_2 递归地调用上面两步，直到满足停止条件为止。

❍ 将输入空间划分为 M 个区域 $R_1,R_2,\cdots,R_m$，生成决策树：

$$f(\vec{\boldsymbol{x}})=\sum_{m=1}^{M}\widehat{c_m}I(\vec{\boldsymbol{x}}\in R_m)$$

通常的停止条件为下列条件之一：

❑ 节点中样本个数小于预定值；

❑ 样本集的平方误差小于预定值；

❑ 没有更多的特征。

CART 分类树

假设有 K 个分类，样本点属于第 k 类的概率为 $p_k=P(Y=c_k)$。定义概率分布的基尼指数为：

$$Gini(p)=\sum_{k=1}^{K}p_k(1-p_k)=1-\sum_{k=1}^{K}p_k^2$$

对于给定的样本集合 D，设属于类 c_k 的样本子集为 C_k，则基尼指数为：

$$Gini(D)=1-\sum_{k=1}^{K}(\frac{|C_k|}{|D|})^2$$

给定特征 A，根据其是否取某一个可能值 a，样本集 D 被分为两个子集: D_1 和 D_2，其中：

$$D_1=\{(\vec{\boldsymbol{x}},y)\in D\mid\vec{\boldsymbol{x}}^{(A)}=a\}$$
$$D_2=\{(\vec{\boldsymbol{x}},y)\in D\mid\vec{\boldsymbol{x}}^{(A)}\neq a\}=D-D_1$$

定义 $Gini(D,A)$：

$$Gini(D,A)=\frac{|D_1|}{|D|}Gini(D_1)+\frac{|D_2|}{|D|}Gini(D_2)$$

它表示在特征 A 的条件下，集合 D 的基尼指数。

对于最简单的二项分布，设 $P(X=1)=p, P(X=0)=1-p$，则其基尼系数与熵的图形如图 2.2 所示。可以看到基尼系数与熵一样，也是用于度量不确定性的。对于样本集 D，$Gini(D)$ 越小说明样本越属于同一类。

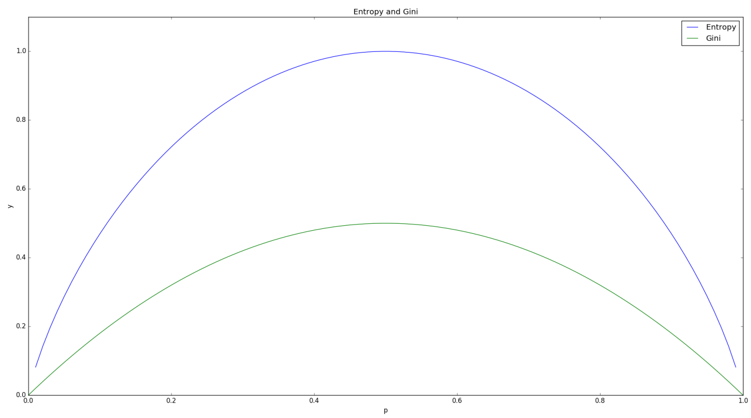


图 2.2　entropy_and_gini

CART 分类树采用基尼指数选择最优特征。CART 分类树的生成算法如下。

- 输入
  - 训练数据集 D。
  - 停止计算条件。
- 输出：CART决策树。
- 算法步骤
  - 对每个特征 A，以及它可能的每个值 a，计算 $Gini(D,A)$。
  - 选取最优特征和最优切分点：在所有特征 A 以及所有的切分点 a 中，基尼指数最小的 A 和 a 就是最优特征和最优切分点。根据最优特征和最优切分点将训练集 D 切分成两个子节点。
  - 对两个子节点递归调用上面两步，直到满足停止条件为止。
  - 最终生成 CART 决策树。

通常的停止条件为下列条件之一：

- 节点中样本个数小于预定值；
- 样本集的基尼指数小于预定值；
- 没有更多的特征。

CART 剪枝

CART剪枝是从生成树开始剪掉一些子树，使得决策树变小。剪枝过程由两步组成（假设初始的生成树为 T_0）：

- 从 T_0 开始不断地剪枝，直到剪成一棵单节点的树。这些剪枝树形成一个剪枝树序列 $\{T_0, T_1, \cdots, T_n\}$；
- 从这个剪枝树序列中挑选出最优剪枝树。方法是：通过交叉验证法使用验证数据集对剪枝树序列进行测试。

给出决策树的损失函数为：$C_\alpha(T) = C(T) + \alpha|T|$。其中 $C(T)$ 为决策树对训练数据的预测误差；$|T|$ 为决策树的叶节点个数。

对固定的 α ，存在使 $C_\alpha(T)$ 最小的树。令其为 T_α，可以证明 T_α 是唯一的。

- 当 α 大时，T_α 偏小（即决策树比较简单）。
- 当 α 小时，T_α 偏大（即决策树比较复杂）。
- 当 $\alpha = 0$ 时，生成树就是最优的；
- 当 $\alpha = \infty$ 时，根组成的一个单节点树就是最优的。

考虑生成树 T_0。对 T_0 内任意节点 t，以 t 为单节点树（记作 $\tilde{t}$）的损失函数为：$C_\alpha(\tilde{t}) = C(\tilde{t}) + \alpha$，以 t 为根的子树 T_t 的损失函数为：$C_\alpha(T_t) = C(T_t) + \alpha|T_t|$。可以证明：

- 当 $\alpha = 0$ 及充分小时，有 $C_\alpha(T_t) < C_\alpha(\tilde{t})$；
- 当 α 增大到某个值时，有 $C_\alpha(T_t) = C_\alpha(\tilde{t})$；
- 当 α 再增大时，有 $C_\alpha(T_t) > C_\alpha(\tilde{t})$。

因此令 $\alpha = \frac{C(\tilde{t}) - C(T_t)}{|T_t| - 1}$，此时 T_t 与 $\tilde{t}$ 有相同的损失函数值，但是 $\tilde{t}$ 的叶节点更少。于是对 T_t 进行剪枝成一棵单节点树 $\tilde{t}$ 了。

对 T_0 内部的每一个节点 t ，定义 $g(t) = \frac{C(t) - C(T_t)}{|T_t| - 1}$ 。设 T_0 内 $g(t)$ 最小的子树为 T_t^*，令该最小值的 $g(t)$ 为 $\tilde{\alpha}_1$。从 T_0 中剪去 T_t^*，即得到剪枝树 T_1。重复这种“求 $g(t)$ - 剪枝”过程，直到根节点即完成剪枝过程。在此过程中不断增加 $\tilde{\alpha}_i$ 的值，从而生成剪枝树序列。

CART剪枝交叉验证过程是通过验证数据集来测试剪枝树序列 $\{T_0, T_1, \cdots, T_n\}$ 中各剪枝树的。对于CART回归树，是考察剪枝树的平方误差，平方误差最小的决策树被认为是最优决策树。对于CART分类树，是考察剪枝树的基尼指数，基尼指数最小的决策树被认为是最优决策树。

CART剪枝算法的描述如下。

- 输入：CART生成树 T_0。
- 输出：CART剪枝树 T_α。
- 算法步骤
  - 令 $k = 0, T = T_0, \alpha = \infty$。

- 自下而上地对树 T 各内部节点 t 计算 $g(t)=\frac{C(t)-C(T_t)}{|T_t|-1}$。
- 对所有的内部节点，$\tilde{\alpha}_{k+1}=\min_t(g(t))$，令 $t^*=\arg\min_t(g(t))$。对内部节点 t^* 进行剪枝得到树 T_{k+1}。
- 令 $T=T_{k+1};k=k+1$。
- 若 T 不是由根节点单独构成的树，则继续前面的步骤。
- 采用交叉验证法在剪枝树序列 $T_0,T_1,\cdots,T_n$ 中选取最优剪枝树 T_α。

2.2.4 连续值和缺失值的处理

现实中应用决策树的一个常见困难是：学习任务中常常会遇到连续特征，如个人身高、体重等特征取值就是连续值。可以通过采用二分法bi-partition对连续特征进行离散化处理。给定样本集 D 和连续特征 A，假设该特征在 D 上的取值从小到大进行排列为 $a_1,a_2,\cdots,a_M$。可以选取 $M-1$ 个划分点，依次为：$\frac{a_1+a_2}{2},\frac{a_2+a_3}{2},\cdots,\frac{a_{M-1}+a_M}{2}$。然后就可以像离散特征一样来考察这些划分点，选取最优的划分点进行样本集和的划分。这也是 C4.5 算法采取的方案。

现实中应用决策树的另外一个常见困难是：学习任务中遇到不完整样本，即某些样本的某些特征的取值缺失。如果简单地丢掉这些不完整的样本很可能会浪费大量有效的信息。

给定训练集 D 和特征 A，令 $\tilde{D}$ 表示 D 中在特征 A 上没有缺失的样本子集。假定特征 A 有 M 个可取值 $a_1,a_2,\cdots,a_M$，令 $\tilde{D}^i$ 表示 $\tilde{D}$ 中在特征 A 上取值为 a_i 的样本的子集；$\tilde{D}_k$ 表示 $\tilde{D}$ 中属于第 k 类的样本子集（一共有 K 个分类），则有：

$$\tilde{D}=\bigcup_{k=1}^{K}\tilde{D}_k=\bigcup_{i=1}^{M}\tilde{D}^i$$

假定为每个样本 $\vec{\boldsymbol{x}}$ 赋予一个权重 $w_{\vec{\boldsymbol{x}}}$，定义：

$$\rho=\frac{\sum_{\vec{\boldsymbol{x}}\in\tilde{D}}w_{\vec{\boldsymbol{x}}}}{\sum_{\vec{\boldsymbol{x}}\in D}w_{\vec{\boldsymbol{x}}}}$$
$$\tilde{p}_k=\frac{\sum_{\vec{\boldsymbol{x}}\in\tilde{D}_k}w_{\vec{\boldsymbol{x}}}}{\sum_{\vec{\boldsymbol{x}}\in\tilde{D}}w_{\vec{\boldsymbol{x}}}},k=1,2,\cdots,K$$
$$\tilde{r}_i=\frac{\sum_{\vec{\boldsymbol{x}}\in\tilde{D}^i}w_{\vec{\boldsymbol{x}}}}{\sum_{\vec{\boldsymbol{x}}\in\tilde{D}}w_{\vec{\boldsymbol{x}}}},i=1,2,\cdots,M$$

其物理意义如下。

- ρ：表示无缺失值样本占总体样本的比例。
- $\tilde{p}_k$：表示无缺失值样本中，第 k 类所占的比例。
- $\tilde{r}_i$：表示无缺失值样本中，在特征 A 上取值为 a_i 的样本所占的比例。

于是可以将信息增益的计算公式修正为：

$$g(D,A)=\rho\times g(\tilde{D},A)=\rho\times\left(H(\tilde{D})-\sum_{i=1}^{M}\tilde{r}_i H(\tilde{D}^i)\right)$$

其中，$H(\tilde{D})=-\sum_{k=1}^{K}\tilde{p}_k\log\tilde{p}_k$。

在通过特征 A 划分样本 $\vec{\boldsymbol{x}}$ 时，让它以不同的概率分散到不同的子节点中去：

- 如果样本在划分特征上的取值已知，则将它划入与其对应的子节点，且权值在子节点中保持为 $w_{\vec{\boldsymbol{x}}}$；
- 如果样本在划分特征上的取值缺失，则将它同时划入所有的子节点，且在子节点中该样本的权值进行调整：在特征取值为 a_i 对应的子节点中，该样本的权值调整为 $\tilde{r}_i\times w_{\vec{\boldsymbol{x}}}$。

2.3 Python 实战

scikit-learn 中有两类决策树，它们均采用优化的CART决策树算法。

2.3.1 回归决策树（DecisionTreeRegressor）

DecisionTreeRegressor实现了回归决策树，用于回归问题。它的原型为：

```
class sklearn.tree.DecisionTreeRegressor(criterion='mse', splitter='best',
    max_depth=None, min_samples_split=2, min_samples_leaf=1,
     min_weight_fraction_leaf=0.0, max_features=None,
    random_state=None, max_leaf_nodes=None, presort=False)
```

参数

- criterion：一个字符串，指定切分质量的评价准则。默认为'mse'，且只支持该字符串，表示均方误差。
- splitter：一个字符串，指定切分原则，可以为如下。
  - 'best'：表示选择最优的切分。
  - 'random'：表示随机切分。
- max_features：可以为整数、浮点、字符串或者None，指定寻找best split时考虑的特征数量。
  - 如果是整数，则每次切分只考虑 max_features 个特征。
  - 如果是浮点数，则每次切分只考虑 max_features * n_features 个特征（max_features 指定了百分比）。

- ❍ 如果是字符串 'auto'或者'sqrt'，则 max_features等于 n_features。
- ❍ 如果是字符串'log2'，则 max_features等于 log2(n_features)。
- ❍ 如果是 None，则 max_features等于 n_features。

> 如果已经考虑了max_features 个特征，但是还没有找到一个有效的切分，那么还会继续寻找下一个特征，直到找到一个有效的切分为止。

- ❑ max_depth：可以为整数或者None，指定树的最大深度。
  - ❍ 如果为None，则表示树的深度不限（直到每个叶子都是纯的，即叶节点中所有样本点都属于一个类，或者叶子中包含小于 min_samples_split 个样本点）；
  - ❍ 如果 max_leaf_nodes参数非 None，则忽略此选项。
- ❑ min_samples_split：为整数，指定每个内部节点（非叶节点）包含的最少的样本数。
- ❑ min_samples_leaf：为整数，指定每个叶节点包含的最少的样本数。
- ❑ min_weight_fraction_leaf：为浮点数，叶节点中样本的最小权重系数。
- ❑ max_leaf_nodes：为整数或者None，指定叶节点的最大数量。
  - ❍ 如果为None，此时叶节点数量不限；
  - ❍ 如果非None，则max_depth被忽略。
- ❑ class_weight：为一个字典、字典的列表、字符串'balanced'或者 None，它指定了分类的权重。权重的形式为：{class_label:weight}。
  - ❍ 如果为None，则每个分类的权重都为 1。
  - ❍ 字符串'balanced' 表示分类的权重是样本中各分类出现的频率的反比。

> 如果提供了 sample_weight参数（由 fit方法提供），则这些权重都会乘以sample_weight。

- ❑ random_state：一个整数或者一个RandomState实例，或者None。
  - ❍ 如果为整数，则它指定了随机数生成器的种子。
  - ❍ 如果为RandomState实例，则指定了随机数生成器。
  - ❍ 如果为None，则使用默认的随机数生成器。
- ❑ presort：一个布尔值，指定是否要提前排序数据从而加速寻找最优切分的过程。设置为 True时，对于大数据集会减慢总体的训练过程；但是对于一个小数据集或者设定了最大深度的情况下，则会加速训练过程。

属性有如下 5 个。

- ❑ feature_importances_：给出了特征的重要程度。该值越高，则该特征越重要（也称为 Gini importance）。
- ❑ max_features_：max_features的推断值。
- ❑ n_features_：当执行fit之后，特征的数量。
- ❑ n_outputs_：当执行fit之后，输出的数量。

❑ tree_：一个 Tree对象，即底层的决策树。

方法有以下 3 种。

❑ fit(X, y[, sample_weight, check_input, ...])：训练模型。
❑ predict(X[, check_input])：用模型进行预测，返回预测值。
❑ score(X,y[,sample_weight])：返回预测性能得分。设预测集为 T_{test}，真实值为 y_i，真实值的均值为 $\bar{y}$，预测值为 $\hat{y}_i$，则：

$$score = 1 - \frac{\sum_{T_{test}}(y_i - \hat{y}_i)^2}{(y_i - \bar{y})^2}$$

❍ score不超过 1，但是可能为负值（预测效果太差）。
❍ score越大，预测性能越好。

下面给出一个示例。首先导入包：

```
import numpy as np
from sklearn.tree import DecisionTreeRegressor
from sklearn import cross_validation
import matplotlib.pyplot as plt
```

给出一个随机产生的数据集：

```
def creat_data(n):
    np.random.seed(0)
    X = 5 * np.random.rand(n, 1)
    y = np.sin(X).ravel()
    noise_num=(int)(n/5)
    y[::5] += 3 * (0.5 - np.random.rand(noise_num))
    return cross_validation.train_test_split(X, y,
        test_size=0.25,random_state=1)
```

❑ 参数：n为数据集容量。
❑ 返回值：返回一个元组，元素依次为：训练样本集、测试样本集、训练样本集对应的值、测试样本集对应的值。

creat_data函数产生的数据集是在 sin(x) 函数基础之上添加了若干个随机噪声产生的。x 是随机在 0 ~ 1 之间产生的，y 是 sin(x)，其中 y 每隔 5 个点添加一个随机噪声。然后将数据集随机切分成训练集和测试集。指定测试集样本大小为原样本集的 0.25 倍。

然后给出测试函数：

```
def test_DecisionTreeRegressor(*data):
    X_train,X_test,y_train,y_test=data
    regr = DecisionTreeRegressor()
    regr.fit(X_train, y_train)
```

```
print("Training score:%f"%(regr.score(X_train,y_train)))
print("Testing score:%f"%(regr.score(X_test,y_test)))
##绘图
fig=plt.figure()
ax=fig.add_subplot(1,1,1)
X = np.arange(0.0, 5.0, 0.01)[:, np.newaxis]
Y = regr.predict(X)
ax.scatter(X_train, y_train, label="train sample",c='g')
ax.scatter(X_test, y_test, label="test sample",c='r')
ax.plot(X, Y, label="predict_value", linewidth=2,alpha=0.5)
ax.set_xlabel("data")
ax.set_ylabel("target")
ax.set_title("Decision Tree Regression")
ax.legend(framealpha=0.5)
plt.show()
```

在test_DecisionTreeRegressor函数中，给出了对 x 上每个点的预测值（考虑到连续值有无穷多，采取的方式是 [0,5]之间，步长为 0.01 ）。DecisionTreeRegressor函数的图形如图 2.3 所示。使用该函数：

```
X_train,X_test,y_train,y_test=creat_data(100)
test_DecisionTreeRegressor(X_train,X_test,y_train,y_test)
```

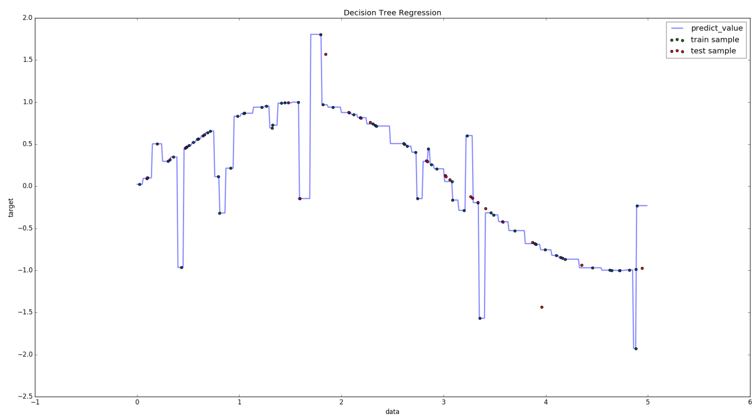


图 2.3　DecisionTreeRegressor

运行结果如下：

```
Training score:1.000000
Testing score:0.789107
```

可以看到对于训练样本的拟合相当好，但是对于测试样本的拟合就差强人意。

接下来，检验随机划分与最优划分的影响，给出函数：

```
def test_DecisionTreeRegressor_splitter(*data):
    X_train,X_test,y_train,y_test=data
    splitters=['best','random']
    for splitter in splitters:
        regr = DecisionTreeRegressor(splitter=splitter)
        regr.fit(X_train, y_train)
        print("Splitter %s"%splitter)
        print("Training score:%f"%(regr.score(X_train,y_train)))
        print("Testing score:%f"%(regr.score(X_test,y_test)))
```

使用该函数：

```
X_train,X_test,y_train,y_test=creat_data(100)
test_DecisionTreeRegressor_splitter(X_train,X_test,y_train,y_test)
```

运行结果如下：

```
Splitter best
Training score:1.000000
Testing score:0.789107
Splitter random
Training score:1.000000
Testing score:0.778989
```

可以看到对于本问题，最优划分预测性能较强，但是相差不大。而对于训练集的拟合，二者都拟合得相当好。

最后考察决策树深度的影响。决策树的深度对应着树的复杂度。决策树越深，则模型越复杂，给出函数：

```
def test_DecisionTreeRegressor_depth(*data,maxdepth):
    X_train,X_test,y_train,y_test=data
    depths=np.arange(1,maxdepth)
    training_scores=[]
    testing_scores=[]
    for depth in depths:
        regr = DecisionTreeRegressor(max_depth=depth)
        regr.fit(X_train, y_train)
        training_scores.append(regr.score(X_train,y_train))
        testing_scores.append(regr.score(X_test,y_test))

    ## 绘图
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    ax.plot(depths,training_scores,label="traing score")
    ax.plot(depths,testing_scores,label="testing score")
```

```
    ax.set_xlabel("maxdepth")
    ax.set_ylabel("score")
    ax.set_title("Decision Tree Regression")
    ax.legend(framealpha=0.5)
    plt.show()
```

使用该函数：

```
X_train,X_test,y_train,y_test=creat_data(100)
test_DecisionTreeRegressor_depth(X_train,X_test,y_train,y_test,maxdepth=20)
```

运行结果如图 2.4 所示。可以看到随着树深度的加深（对应着模型复杂度的提高），模型对训练集和预测集的拟合都在提高。由于样本只有 100 个，因此理论上二叉树最深为 $\log_2(100)=6.65$。即树深度为 7 之后，再也无法划分了（每个子节点都只有一个节点了）。

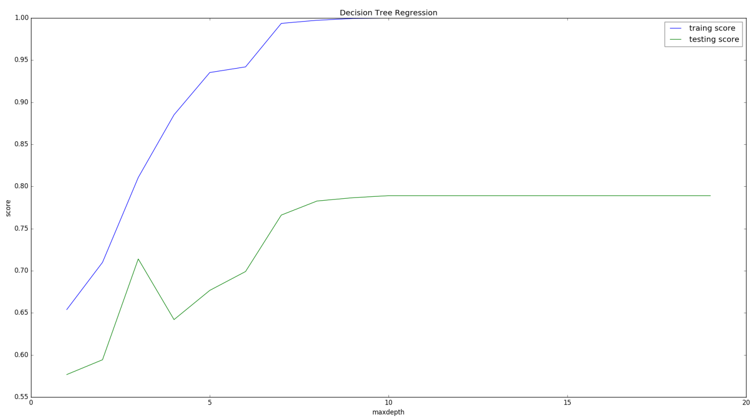


图 2.4　DecisionTreeRegressor_depth

不同深度的决策树如图 2.5 所示。可以看到，深度越小的决策树越简单，它将特征空间划分的折线越少。深度越深的决策树越复杂，它将特征空间划分的折线越多（也就是越曲折）。

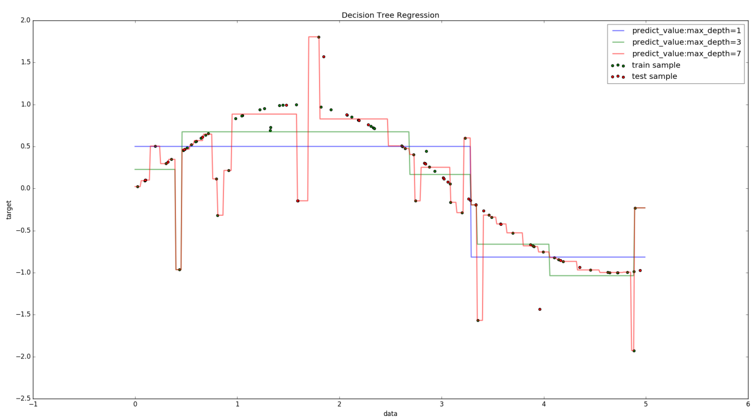


图 2.5　DecisionTreeRegressor_depth_plot

2.3.2 分类决策树（DecisionTreeClassifier）

DecisionTreeClassifier实现了分类决策树，用于分类问题。它的原型为：

```
sklearn.tree.DecisionTreeClassifier(criterion='gini', splitter='best', max_depth=None,
min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0,
max_features=None,random_state=None, max_leaf_nodes=None, class_weight=None,
presort=False)
```

参数如下。

- criterion：一个字符串，指定切分质量的评价准则。可以为如下。
  - 'gini'：表示切分时评价准则是Gini系数。
  - 'entropy'：表示切分时评价准则是熵。
- splitter：一个字符串，指定切分原则，可以为如下。
  - 'best'：表示选择最优的切分。
  - 'random'：表示随机切分。
- max_features：可以为整数、浮点、字符串或者None，指定了寻找best split时考虑的特征数量。
  - 如果是整数，则每次切分只考虑 max_features 个特征。
  - 如果是浮点数，则每次切分只考虑 max_features * n_features 个特征（max_features 指定了百分比）。
  - 如果是字符串 'auto'或者'sqrt'，则 max_features等于 sqrt(n_features)。
  - 如果是字符串'log2'，则 max_features等于 log2(n_features)。
  - 如果是 None，则 max_features等于 n_features。

> 如果已经考虑了max_features 个特征，但是还没有找到一个有效的切分，那么还会继续寻找下一个特征，直到找到一个有效的切分为止。

- max_depth：可以为整数或者None，指定树的最大深度。
  - 如果为None，则表示树的深度不限（直到每个叶子都是纯的，即叶节点中所有的样本点都属于一个类，或者叶子中包含小于 min_samples_split 个样本点）;
  - 如果 max_leaf_nodes 非 None，则忽略此选项。
- min_samples_split：为整数，指定每个内部节点（非叶节点）包含的最少的样本数。
- min_samples_leaf：为整数，指定每个叶节点包含的最少的样本数。
- min_weight_fraction_leaf：为浮点数，叶节点中样本的最小权重系数。
- max_leaf_nodes：为整数或者None，指定最大的叶节点数量。
  - 如果为None，此时叶节点数量不限。
  - 如果非None，则max_depth被忽略。

- class_weight：为一个字典、字典的列表、字符串'balanced'，或者 None，它指定了分类的权重。权重的形式为：{class_label:weight}。
  - 如果为None，则每个分类的权重都为 1。
  - 字符串'balanced' 表示分类的权重是样本中各分类出现的频率的反比。

> 如果 sample_weight提供了权重（由 fit方法提供），则这些权重都会乘以sample_weight。

- random_state：一个整数或者一个RandomState实例，或者None。
  - 如果为整数，则它指定了随机数生成器的种子。
  - 如果为RandomState实例，则指定了随机数生成器。
  - 如果为None，则使用默认的随机数生成器。
- presort：一个布尔值，指定是否要提前排序数据，从而加速寻找最优切分的过程。设置为 True时，对于大数据集会减慢总体的训练过程；但是对于一个小数据集或者设定了最大深度的情况下，会加速训练过程。

属性有以下 7 个。

- classes_：分类的标签值。
- feature_importances_：给出了特征的重要程度。该值越高，则该特征越重要（也称为 Gini importance）。
- max_features_：max_features的推断值。
- n_classes_：给出了分类的数量。
- n_features_：执行fit之后，特征的数量。
- n_outputs_：执行fit之后，输出的数量。
- tree_：一个 Tree对象，即底层的决策树。

方法有以下 5 种。

- fit(X, y[, sample_weight, check_input, ...])：训练模型。
- predict(X[, check_input])：用模型进行预测，返回预测值。
- predict_log_proba(X)：返回一个数组，数组的元素依次是X预测为各个类别的概率的对数值。
- predict_proba(X)：返回一个数组，数组的元素依次是X预测为各个类别的概率值。
- score(X,y[,sample_weight])：返回在 (X,y)上预测的准确率 (accuracy)。

下面给出一个示例。首先导入包：

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import  datasets
from sklearn.tree import DecisionTreeClassifier
from sklearn import cross_validation
```

采用鸢尾花数据集。鸢尾花数据集一共有 150 个数据，这些数据分为 3 类（分别为setosa, versicolor, virginica），每类 50 个数据。每个数据包含 4 个属性：萼片 (sepal) 长度、萼片宽度、花瓣 (petal) 长度、花瓣宽度。这里给出load_data函数：

```
def load_data():
    iris=datasets.load_iris()
    X_train=iris.data
    y_train=iris.target
    return cross_validation.train_test_split(X_train, y_train,test_size=0.25,
        random_state=0,stratify=y_train)
```

在这里采用分层采样。因为原始数据集中，前 50 个样本都是类别 0，中间 50 个样本都是类别 1，最后 50 个类别都是类别 2。如果不采取分层采用，那么最后切分得到的测试数据集就不是无偏的了。

然后，我们给出了使用DecisionTreeClassifier进行分类的函数：

```
def test_DecisionTreeClassifier(*data):
    X_train,X_test,y_train,y_test=data
    clf = DecisionTreeClassifier()
    clf.fit(X_train, y_train)

    print("Training score:%f"%(clf.score(X_train,y_train)))
    print("Testing score:%f"%(clf.score(X_test,y_test)))
```

调用test_DecisionTreeClassifier：

```
X_train,X_test,y_train,y_test=load_data()
test_DecisionTreeClassifier(X_train,X_test,y_train,y_test)
```

执行结果如下：

```
Training score:1.000000
Testing score:0.974359
```

可以看到对训练数据集完全拟合，对测试数据集拟合精度高达 97.4359%。

现在考察评价切分质量的评价准则criterion对于分类性能的影响，给出函数：

```
def test_DecisionTreeClassifier_criterion(*data):
    X_train,X_test,y_train,y_test=data
    criterions=['gini','entropy']
    for criterion in criterions:
        clf = DecisionTreeClassifier(criterion=criterion)
        clf.fit(X_train, y_train)
        print("criterion:%s"%criterion)
        print("Training score:%f"%(clf.score(X_train,y_train)))
        print("Testing score:%f"%(clf.score(X_test,y_test)))
```

使用该函数：

```
X_train,X_test,y_train,y_test=creat_data(100)
test_DecisionTreeClassifier_criterion(X_train,X_test,y_train,y_test)
```

运行结果如下：

```
criterion:gini
Training score:1.000000
Testing score:0.974359
criterion:entropy
Training score:1.000000
Testing score:0.948718
```

可以看到对于本问题二者对于训练集的拟合都非常完美（100%），对于测试集的预测都较高，但是稍有不同；使用Gini系数的策略预测性能较高。

接下来，检验随机划分与最优划分的影响，给出函数：

```
def test_DecisionTreeClassifier_splitter(*data):
    X_train,X_test,y_train,y_test=data
    splitters=['best','random']
    for splitter in splitters:
        clf = DecisionTreeClassifier(splitter=splitter)
        clf.fit(X_train, y_train)
        print("splitter:%s"%splitter)
        print("Training score:%f"%(clf.score(X_train,y_train)))
        print("Testing score:%f"%(clf.score(X_test,y_test)))
```

使用该函数：

```
X_train,X_test,y_train,y_test=creat_data(100)
test_DecisionTreeClassifier_splitter(X_train,X_test,y_train,y_test)
```

运行结果如下：

```
splitter:best
Training score:1.000000
Testing score:0.974359
splitter:random
Training score:1.000000
Testing score:0.948718
```

可以看到对于本问题二者对于训练集的拟合都非常完美（100%），对于测试集的预测都较高，但是稍有不同；使用最优划分的性能要高于随机划分。

最后考察决策树深度的影响。决策树的深度对应着树的复杂度。决策树越深，则模型越复杂，给出函数：

```
def test_DecisionTreeClassifier_depth(*data,maxdepth):
    X_train,X_test,y_train,y_test=data
    depths=np.arange(1,maxdepth)
    training_scores=[]
    testing_scores=[]
    for depth in depths:
        clf = DecisionTreeClassifier(max_depth=depth)
        clf.fit(X_train, y_train)
        training_scores.append(clf.score(X_train,y_train))
        testing_scores.append(clf.score(X_test,y_test))

    ## 绘图
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    ax.plot(depths,training_scores,label="traing score",marker='o')
    ax.plot(depths,testing_scores,label="testing score",marker='*')
    ax.set_xlabel("maxdepth")
    ax.set_ylabel("score")
    ax.set_title("Decision Tree Classification")
    ax.legend(framealpha=0.5,loc='best')
    plt.show()
```

使用该函数:

```
X_train,X_test,y_train,y_test=creat_data(100)
test_DecisionTreeClassifier_depth(X_train,X_test,y_train,y_test,maxdepth=100)
```

运行结果如图 2.6 所示。可以看到随着树深度的增加（对应着模型复杂度的提高），模型对训练集和预测集的拟合都在提高。这里训练数据集大小仅为 150，不考虑任何条件，只需要一棵深度为 $\log_2 150 \leqslant 8$ 的二叉树就能够完全拟合数据，使得每个叶子节点最多只有一个样本。考虑到决策树算法中的提前终止条件（比如叶子节点中所有样本都是同一类则不再划分，此时叶节点中有超过一个样本），则树的深度小于 8。

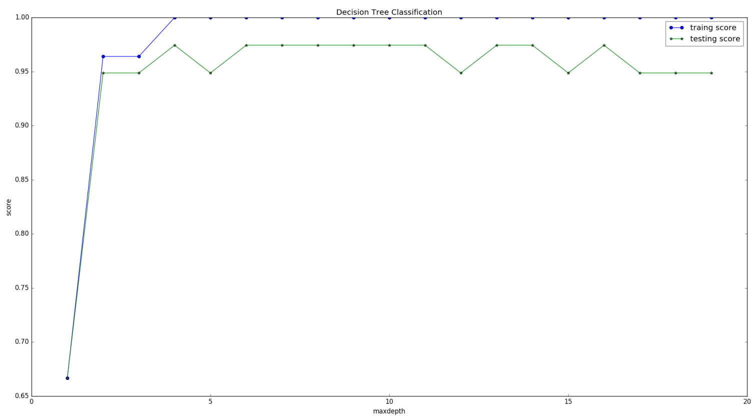


图 2.6 DecisionTreeClassifier_depth

2.3.3 决策图

当训练完毕一棵决策树时，可以通过 `sklearn.tree.export_graphviz(classifier,out_file)` 来将决策树转化成 Graphviz 格式的文件。对上面的`DecisionTreeClassifier`例子，使用 `export_graphviz` 函数如下：

> 这里要求安装Graphviz程序。Graphviz是贝尔实验室开发的一个开源的工具包，用于绘制结构化的图形网络，支持多种格式输出，如常用的图片格式、SVG、PDF 格式等，且支持 Linux/Windows 操作系统。

```
from sklearn.tree import export_graphviz
X_train,X_test,y_train,y_test=load_data()
clf = DecisionTreeClassifier()
clf.fit(X_train, y_train)
export_graphviz(clf,"F:/out")
```

然后通过Graphviz的 dot工具，在命令行中运行命令`dot.exe -Tpdf F:/out -o F:/out.pdf`生成pdf格式的决策图，如图 2.7 所示。或者执行`dot.exe -Tpng F:/out -o F:/out.png`来生成png格式的决策图。其中`-T`选项指定了输出文件的格式，`-o`选项指定了输出文件名。

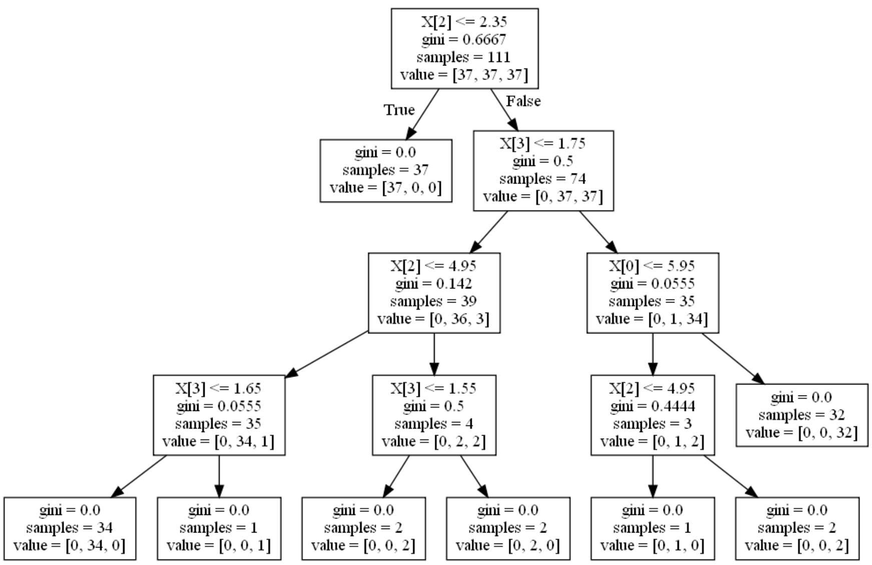


图 2.7 Decision_Graph

第 3 章

贝叶斯分类器

3.1 概述

贝叶斯分类是一种分类算法的总称，这种算法均以贝叶斯定理为基础，故统称为贝叶斯分类。

贝叶斯分类器的分类原理是通过某对象的先验概率，利用贝叶斯公式计算出其后验概率，即该对象属于某一类的概率，选择具有最大后验概率的类作为该对象所属的类。

贝叶斯分类器的主要特点有：

- 属性可以离散，也可以连续；
- 数学基础扎实，分类效率稳定；
- 对缺失和噪声数据不太敏感；
- 属性如果不相关，分类效果很好；如果相关，则不低于决策树。

3.2 算法笔记精华

3.2.1 贝叶斯定理

贝叶斯定理是用数学的方法来解释生活中大家都知道的常识，而机器学习使用的各种算法中，最常见的就是贝叶斯定理。

先验概率是根据以往经验和分析得到的概率。如：你在山洞门口，觉得山洞中有熊出现的事件为 Y ，然后听到山洞中传来一阵熊吼的事件为 X 。一开始你以为山洞中有熊的概率为 $P(Y)$ ，听到熊吼之后认为有熊的概率为 $P(Y/X)$ 。很明显 $P(Y/X) > P(Y)$。这里：

- $P(Y)$ 为先验概率，是根据以往的数据分析或者经验得到的概率；
- $P(Y/X)$ 为后验概率，是得到本次试验的信息从而重新修正的概率。

设 S 为试验 E 的样本空间。$B_1, B_2, \cdots, B_n$ 为 E 的一组事件。若：

- $B_i \bigcap B_j = \phi, i \neq j, i, j = 1, 2, \cdots, n$
- $B_1 \bigcup B_2 \bigcup \cdots \bigcup B_n = S$

则称 $B_1, B_2, \cdots, B_n$ 为样本空间 S 的一个划分。如果 $B_1, B_2, \cdots, B_n$ 为样本空间 S 的一个划分，则对于每次试验，事件 $B_1, B_2, \cdots, B_n$ 中有且仅有一个事件发生。

全概率公式：设试验 E 的样本空间为 S，A 为 E 的事件，$B_1, B_2, \cdots, B_n$ 为样本空间 S 的一个划分，且 $P(B_i) \geqslant 0 (i = 1, 2, \cdots, n)$，则有：

$$P(A) = P(A/B_1)P(B_1) + P(A/B_2)P(B_2) + \cdots + P(A/B_n)P(B_n) = \sum_{j=1}^{n} P(A/B_j)P(B_j)$$

贝叶斯定理：设试验 E 的样本空间为 S ，A 为 E 的事件，$B_1, B_2, \cdots, B_n$ 为样本空间 S 的一个划分，且 $P(A) > 0, P(B_i) \geqslant 0 (i = 1, 2, \cdots, n)$ ，则有：

$$P(B_i/A) = \frac{P(A/B_i)P(B_i)}{\sum_{j=1}^{n} P(A/B_j)P(B_j)}$$

3.2.2 朴素贝叶斯法

原理

设样本 $\vec{\boldsymbol{x}} = (x^{(1)}, x^{(2)}, \cdots, x^{(n)})^T \in \mathcal{X} \subseteq \mathbb{R}^n$，设标记 $y \in \mathcal{Y} = \{c_1, c_2, \cdots, c_K\}$。令 X 为 $\mathcal{X}$ 上的随机向量，Y 为 $\mathcal{Y}$ 上的随机变量，$P(X, Y)$ 为 X 和 Y 的联合概率分布。假定训练数据集 $T = \{(\vec{\boldsymbol{x}}_1, y_1), (\vec{\boldsymbol{x}}_2, y_2), \cdots, (\vec{\boldsymbol{x}}_N, y_N)\}$ 由 $P(X, Y)$ 独立同分布产生，那么朴素贝叶斯法可从训练数据集中学习联合概率分布 $P(X, Y)$，也就是学习下列概率分布。

- 先验概率分布：$P(Y = c_k), k = 1, 2, \cdots, K$。
- 条件概率分布：$P(X = \vec{\boldsymbol{x}}/Y = c_k), k = 1, 2, \cdots, K$。

朴素贝叶斯法假设：在分类确定的条件下，用于分类的特征是条件独立的。即：

$$\begin{aligned}
P(X = \vec{\boldsymbol{x}}/Y = c_k) &= P(X^{(1)} = x^{(1)}, X^{(2)} = x^{(2)}, \cdots, X^{(n)} = x^{(n)}/Y = c_k) \\
&= \prod_{j=1}^{n} P(X^{(j)} = x^{(j)}/Y = c_k), k = 1, 2, \cdots, K
\end{aligned}$$

根据贝叶斯定理：

$$P(Y=c_k/X=\vec{\boldsymbol{x}})=\frac{P(X=\vec{\boldsymbol{x}}/Y=c_k)P(Y=c_k)}{\sum_{j=1}^{K}P(X=\vec{\boldsymbol{x}}/Y=c_j)P(Y=c_j)}$$

考虑分类特征的条件独立假设有：

$$P(Y=c_k/X=\vec{\boldsymbol{x}})=\frac{P(Y=c_k)\prod_{i=1}^{n}P(X^{(i)}=x^{(i)}/Y=c_k)}{\sum_{j=1}^{K}P(X=\vec{\boldsymbol{x}}/Y=c_j)P(Y=c_j)},k=1,2,\cdots,K$$

于是朴素贝叶斯分类器表示为：

$$y=f(\vec{\boldsymbol{x}})=\arg\max_{c_k}\frac{P(Y=c_k)\prod_{i=1}^{n}P(X^{(i)}=x^{(i)}/Y=c_k)}{\sum_{j=1}^{K}P(X=\vec{\boldsymbol{x}}/Y=c_j)P(Y=c_j)}$$

由于对所有的 $c_k,k=1,2,\cdots,K$ ，上式的分母都相同（均为 $P(\vec{\boldsymbol{x}})$），因此上式可重写为：

$$y=f(\vec{\boldsymbol{x}})=\arg\max_{c_k}P(Y=c_k)\prod_{i=1}^{n}P(X^{(i)}=x^{(i)}/Y=c_k)$$

朴素贝叶斯法的学习

在朴素贝叶斯法中，要学习的参数就是以下两种概率：

- ❑ 先验概率 $P(Y=c_k)$
- ❑ 条件概率 $P(X^{(j)}=x^{(j)}/Y=c_k)$

通常采用极大似然估计这两种概率。

- ❑ 先验概率 $P(Y=c_k)$ 的极大似然估计为：

$$P(Y=c_k)=\frac{1}{N}\sum_{i=1}^{N}I(y_i=c_k),k=1,2,\cdots,K$$

- ❑ 条件概率 $P(X^{(j)}=a_{j\,l}/Y=c_k)$ 的极大似然估计为：

$$P(X^{(j)}=a_{j\,l}/Y=c_k)=\frac{\sum_{i=1}^{N}I(x_i^{(j)}=a_{j\,l},y_i=c_k)}{\sum_{i=1}^{N}I(y_i=c_k)}$$

$$j=1,2,\cdots,n;$$

$$l=1,2,\cdots,s_j;$$

$$k=1,2,\cdots,K$$

其中，$a_{j1},a_{j2},\cdots,a_{js_j}$ 为第 j 个特征 $x^{(j)}$ 可能的取值。

朴素贝叶斯法算法

这里给出朴素贝叶斯法算法：

- 输入
  - 训练集 $T=\{(\vec{\boldsymbol{x}}_1,y_1),(\vec{\boldsymbol{x}}_2,y_2),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\}$, $\vec{\boldsymbol{x}}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(n)})^T$, $x_i^{(j)}$ 为第 i 个样本的第 j 个特征，其中 $x_i^{(j)}\in\{a_{j1},a_{j2},\cdots,a_{js_j}\}$，$a_{j\,l}$ 为第 j 个特征可能取到的第 l 个值，$j=1,2,\cdots,n,\quad l=1,2,\cdots,s_j,\quad y_i\in\{c_1,c_2,\cdots,c_K\}$。
  - 实例 $\vec{\boldsymbol{x}}$
- 输出：实例 $\vec{\boldsymbol{x}}$ 的分类。
- 算法步骤
  - 计算先验概率的估计值以及条件概率的估计值：

$$P(Y=c_k)=\frac{\sum_{i=1}^N I(y_i=c_k)}{N},k=1,2,\cdots,K$$
$$P(X^{(j)}=a_{j\,l}/Y=c_k)=\frac{\sum_{i=1}^N I(x_i^{(j)}=a_{j\,l},y_i=c_k)}{\sum_{i=1}^N I(y_i=c_k)}$$
$$j=1,2,\cdots,n;\quad l=1,2,\cdots,s_j;\quad k=1,2,\cdots,K$$

  - 对于给定的实例 $\vec{\boldsymbol{x}}=(x^{(1)},x^{(2)},\cdots,x^{(n)})^T$，计算：

$$P(Y=c_k)\prod_{j=1}^n P(X^{(j)}=x^{(j)}/Y=c_k),k=1,2,\cdots,K$$

  - 计算并返回实例 $\vec{\boldsymbol{x}}$ 的分类 y：

$$y=\arg\max_{c_k} P(Y=c_k)\prod_{j=1}^n P(X^{(j)}=x^{(j)}/Y=c_k)$$

贝叶斯估计

设第 j 个特征 $x^{(j)}$ 可能的取值为 $a_{j1},a_{j2},\cdots,a_{js_j}$，则条件概率 $P(X^{(j)}=a_{j\,l}/Y=c_k)$ 的极大似然估计为：

$$P(X^{(j)}=a_{j\,l}/Y=c_k)=\frac{\sum_{i=1}^N I(x_i^{(j)}=a_{j\,l},y_i=c_k)}{\sum_{i=1}^N I(y_i=c_k)}$$
$$j=1,2,\cdots,n;$$
$$l=1,2,\cdots,s_j;$$
$$k=1,2,\cdots,K$$

用极大似然估计可能会出现分母 $\sum_{i=1}^{N} I(y_i = c_k)$ 为 0 的情况，此时可以采用贝叶斯估计（最大后验估计）：

$$P_\lambda(X^{(j)} = a_{j\,l}/Y = c_k) = \frac{\sum_{i=1}^{N} I(x_i^{(j)} = a_{j\,l}, y_i = c_k) + \lambda}{\sum_{i=1}^{N} I(y_i = c_k) + s_j \lambda}$$

$$j = 1, 2, \cdots, n;$$

$$l = 1, 2, \cdots, s_j;$$

$$k = 1, 2, \cdots, K$$

> 它等价于在 $X^{(j)}$ 的各个取值的频数上赋予了一个正数 λ。

它满足概率分布函数的条件：

$$P_\lambda(X^{(j)} = a_{j\,l}/Y = c_k) > 0, l = 1, 2, \cdots, s_j, \quad k = 1, 2, \cdots, K$$

$$\sum_{l=1}^{s_j} P_\lambda(X^{(j)} = a_{j\,l}/Y = c_k) = 1$$

此时 $P(Y = c_k)$ 的贝叶斯估计调整为:

$$P_\lambda(Y = c_k) = \frac{\sum_{i=1}^{N} I(y_i = c_k) + \lambda}{N + K\lambda}$$

- 当 $\lambda = 0$ 时，为极大似然估计。
- 当 $\lambda = 1$ 时，为拉普拉斯平滑。

3.3 Python 实战

在`scikit`中有多种不同的朴素贝叶斯分类器,它们的区别就在于假设了不同的 $P(X^{(j)}/y = c_k)$ 分布，下面介绍三种常用的朴素贝叶斯分类器。

- `GaussianNB`是高斯贝叶斯分类器。它假设特征的条件概率分布满足高斯分布：

$$P(X^{(j)}/y = c_k) = \frac{1}{\sqrt{2\pi\sigma_k^2}} \exp\left(-\frac{(X^{(j)} - \mu_k)^2}{2\sigma_k^2}\right)$$

- MultinomialNB是多项式贝叶斯分类器。它假设特征的条件概率分布满足多项式分布：

$$P(X^{(j)}=a_{s_j}/y=c_k)=\frac{N_{kj}+\alpha}{N_k+\alpha n}$$

其中，a_{s_j} 表示特征 $X^{(j)}$ 的取值，其取值个数为 s_j 个；$N_k=\sum_{i=1}^{N}I(y_i=c_k)$，表示属于类别 c_k 的样本的数量；$N_{kj}=\sum_{i=1}^{N}I(y_i=c_k,X^{(j)}=a_{s_j})$，表示属于类别 c_k 且特征 $X^{(j)}=a_{s_j}$ 的样本的数量。α 就是前述贝叶斯估计中的 λ。

- BernoulliNB是伯努利贝叶斯分类器。它假设特征的条件概率分布满足二项分布：

$$P(X^{(j)}/y=c_k)=pX^{(j)}+(1-p)(1-X^{(j)})$$

其中，要求特征的取值为 $X^{(j)}\in\{0,1\}$，且 $P(X^{(j)}=1/y=c_k)=p$。

与多项式模型一样，伯努利模型适用于离散特征的情况，所不同的是，伯努利模型中每个特征的取值只能是 1 和 0（以文本分类为例，某个单词在文档中出现过，则其特征值为 1，否则为 0）。

首先导入包：

```
from sklearn import datasets,cross_validation,naive_bayes
import  numpy as np
import  matplotlib.pyplot as plt
```

这里使用的是scikit-learn自带的手写识别数据集Digit Dataset。该数据集由 1797 张样本图片组成，如图 3.1 所示。每张样本图片都是一个 8×8 大小的手写数字位图。为了便于处理，scikit-learn将样本图片转换成 64 维的向量。我们通过下面的函数来观察Digit Dataset数据集：

```
def show_digits():
    digits=datasets.load_digits()
    fig=plt.figure()
    print("vector from images 0:",digits.data[0])
    for i in range(25):
        ax=fig.add_subplot(5,5,i+1)
        ax.imshow(digits.images[i],cmap=plt.cm.gray_r, interpolation='nearest')
    plt.show()
```

调用该函数，结果如下：

```
vector from images 0:
[  0.   0.   5.  13.   9.   1.   0.   0.   0.   0.  13.  15.  10.  15.   5.
   0.   0.   3.  15.   2.   0.  11.   8.   0.   0.   4.  12.   0.   0.   8.
   8.   0.   0.   5.   8.   0.   0.   9.   8.   0.   0.   4.  11.   0.   1.
  12.   7.   0.   0.   2.  14.   5.  10.  12.   0.   0.   0.   0.   6.  13.
  10.   0.   0.   0.]
```

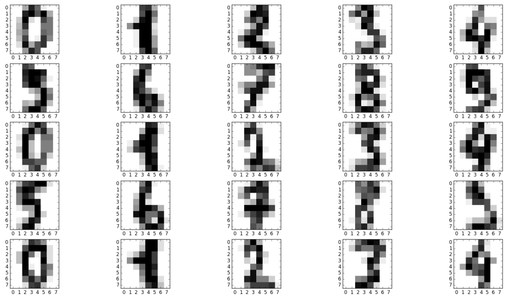


图 3.1 digit_dataset

然后给出加载数据集的函数：

```
def load_data():
    digits=datasets.load_digits()
    return cross_validation.train_test_split(digits.data,digits.target,
        test_size=0.25,random_state=0)
```

- 返回值：一个元组，元组依次是：训练样本集、测试样本集、训练样本集对应的标签值、测试样本集对应的标签值。

在这里采用分层采样。分层采样保证了测试样本集中各类别样本的比例与原始样本集中各类别样本的比例相同。如果不采取分层采用，那么最后切分得到的测试数据集就不是无偏的了。

3.3.1 高斯贝叶斯分类器（GaussianNB）

GaussianNB是高斯贝叶斯分类器，它假设特征的条件概率分布满足高斯分布：

$$P(X^{(j)}/y=c_k)=\frac{1}{\sqrt{2\pi\sigma_k^2}}\exp\left(-\frac{(X^{(j)}-\mu_k)^2}{2\sigma_k^2}\right)$$

其原型为：

```
class sklearn.naive_bayes.GaussianNB
```

GaussianNB没有参数，因此不需要调参。

属性

- class_prior_：一个数组，形状为(n_classes,)，是每个类别的概率 $P(y=c_k)$。
- class_count_：一个数组，形状为(n_classes,),是每个类别包含的训练样本数量。
- theta_：一个数组，形状为(n_classes,n_features)，是每个类别上每个特征的均值 μ_k。
- sigma_：一个数组，形状为(n_classes,n_features)，是每个类别上每个特征的标准差 σ_k。

方法

- ❑ fit(X, y[, sample_weight])：训练模型。
- ❑ partial_fit(X, y[, classes, sample_weight])：追加训练模型。该方法主要用于大规模数据集的训练。此时可以将大数据集划分成若干个小数据集，然后在这些小数据集上连续调用partial_fit方法来训练模型。
- ❑ predict(X)：用模型进行预测，返回预测值。
- ❑ predict_log_proba(X)：返回一个数组，数组的元素依次是X预测为各个类别的概率的对数值。
- ❑ predict_proba(X)：返回一个数组，数组的元素依次是X预测为各个类别的概率值。
- ❑ score(X, y[, sample_weight])：返回在 (X,y)上预测的准确率 (accuracy)。

给出测试高斯贝叶斯分类器的函数：

```
def test_GaussianNB(*data):
    X_train,X_test,y_train,y_test=data
    cls=naive_bayes.GaussianNB()
    cls.fit(X_train,y_train)
    print('Training Score: %.2f' % cls.score(X_train,y_train))
    print('Testing Score: %.2f' % cls.score(X_test, y_test))
```

然后调用test_GaussianNB函数：

```
X_train,X_test,y_train,y_test=load_data()
test_GaussianNB(X_train,X_test,y_train,y_test)
```

运行结果如下：

```
Training Score: 0.86
Testing Score: 0.83
```

可以看到高斯贝叶斯分类器对训练数据集的预测准确率为 86%，对测试数据集的预测准确率为 83%。

3.3.2 多项式贝叶斯分类器（MultinomialNB）

MultinomialNB是多项式贝叶斯分类器，它假设特征的条件概率分布满足多项式分布：

$$P(X^{(j)}=a_{s_j}/y=c_k)=\frac{N_{kj}+\alpha}{N_k+\alpha n}$$

其中，a_{s_j} 表示特征 $X^{(j)}$ 的取值，其取值个数为 s_j 个；$N_k=\sum_{i=1}^{N}I(y_i=c_k)$，表示属于类别 c_k 的样本的数量；$N_{kj}=\sum_{i=1}^{N}I(y_i=c_k,X^{(j)}=a_{s_j})$，表示属于类别 c_k，且特征 $X^{(j)}=a_{s_j}$ 的样本的数量。α 就是前述贝叶斯估计中的 λ。

其原型为：

```
class sklearn.naive_bayes.MultinomialNB(alpha=1.0, fit_prior=True, class_prior=None)
```

参数

- ❑ alpha：一个浮点数，指定 α 值。
- ❑ fit_prior：布尔值。如果为True，则不去学习 $P(y = c_k)$，替代以均匀分布；如果为False, 则去学习 $P(y = c_k)$。
- ❑ class_prior：一个数组。它指定了每个分类的先验概率 $P(y = c_1), P(y = c_2), \cdots, P(y = c_K)$。如果指定了该参数，则每个分类的先验概率不再从数据集中学得。

属性

- ❑ class_log_prior_：一个数组对象，形状为(n_classes,)。给出了每个类别调整后的经验概率分布的对数值。
- ❑ feature_log_prob_: 一个数组对象，形状为(n_classes, n_features)。给出了 $P(X^{(j)}/y = c_k)$ 的经验概率分布的对数值。
- ❑ class_count_：一个数组，形状为(n_classes,), 是每个类别包含的训练样本数量。
- ❑ feature_count_：一个数组，形状为(n_classes, n_features)。训练过程中，每个类别每个特征遇到的样本数。

方法

- ❑ fit(X, y[, sample_weight])：训练模型。
- ❑ partial_fit(X, y[, classes, sample_weight])：追加训练模型。该方法主要用于大规模数据集的训练。此时可以将大数据集划分成若干个小数据集，然后在这些小数据集上连续调用partial_fit方法来训练模型。
- ❑ predict(X)：用模型进行预测，返回预测值。
- ❑ predict_log_proba(X)：返回一个数组，数组的元素依次是X预测为各个类别的概率的对数值。
- ❑ predict_proba(X)：返回一个数组，数组的元素依次是X预测为各个类别的概率值。
- ❑ score(X, y[, sample_weight])：返回在 (X,y)上预测的准确率 (accuracy)。

给出测试多项式贝叶斯分类器的函数：

```
def test_MultinomialNB(*data):
    X_train,X_test,y_train,y_test=data
    cls=naive_bayes.MultinomialNB()
    cls.fit(X_train,y_train)
    print('Training Score: %.2f' % cls.score(X_train,y_train))
    print('Testing Score: %.2f' % cls.score(X_test, y_test))
```

调用test_MultinomialNB函数，运行结果如下：

```
Training Score: 0.91
Testing Score: 0.91
```

可以看到多项式贝叶斯分类器对训练数据集的预测准确率为 91%，对测试数据集的预测准确率为 91%。

接着检验不同的 α 对多项式贝叶斯分类器的预测性能的影响，给出函数：

```
def test_MultinomialNB_alpha(*data):
    X_train,X_test,y_train,y_test=data
    alphas=np.logspace(-2,5,num=200)
    train_scores=[]
    test_scores=[]
    for alpha in alphas:
        cls=naive_bayes.MultinomialNB(alpha=alpha)
        cls.fit(X_train,y_train)
        train_scores.append(cls.score(X_train,y_train))
        test_scores.append(cls.score(X_test, y_test))

    ## 绘图
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    ax.plot(alphas,train_scores,label="Training Score")
    ax.plot(alphas,test_scores,label="Testing Score")
    ax.set_xlabel(r"$\alpha$")
    ax.set_ylabel("score")
    ax.set_ylim(0,1.0)
    ax.set_title("MultinomialNB")
    ax.set_xscale("log")
    plt.show()
```

调用test_MultinomialNB_alpha函数，运行结果如图 3.2 所示。为了便于观察我们将 x 轴设置为对数坐标。可以看到 $\alpha > 100$ 之后，随着 α 的增长，预测准确率在下降。这是因为多项式贝叶斯估计中，假设特征的条件概率分布满足以下多项式分布：

$$P(X^{(j)} = a_{s_j}/y = c_k) = \frac{N_{kj} + \alpha}{N_k + \alpha n}$$

当 $\alpha \to \infty$ 时，$\frac{N_{kj}+\alpha}{N_k+\alpha n} \to \frac{1}{n}$，即对任何类型的特征、该类型特征的任意取值，出现的概率都是 $\frac{1}{n}$。它完全忽略了各个特征之间的差别，也忽略了每个特征内部的分布。在本问题中总的样本数量在 10^3 ，N_k 的量级在 10^2，因此在 $\alpha > 100$ 之后，预测准确率受影响较大。

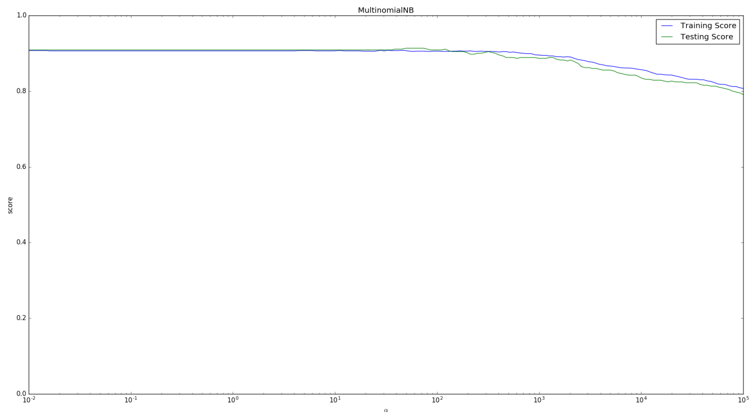


图 3.2 MultinomialNB_alpha

3.3.3 伯努利贝叶斯分类器（BernoulliNB）

BernoulliNB是伯努利贝叶斯分类器。它假设特征的条件概率分布满足二项分布：

$$P(X^{(j)}/y=c_k)=pX^{(j)}+(1-p)(1-X^{(j)})$$

其中，要求特征的取值为 $X^{(j)}\in\{0,1\}$，且 $P(X^{(j)}=1/y=c_k)=p$。

其原型为：

```
class sklearn.naive_bayes.BernoulliNB(alpha=1.0, binarize=0.0, fit_prior=True,
    class_prior=None)
```

参数

- alpha：一个浮点数，指定 α 值。它就是前述贝叶斯估计中的 λ。
- binarize：一个浮点数或者None。
  - 如果为None，那么会假定原始数据已经二元化了。
  - 如果是浮点数，那么会以该数值为界，特征取值大于它的作为 1；特征取值小于它的作为 0。采取这种策略来二元化。
- fit_prior：布尔值。如果为True，则不去学习 $P(y=c_k)$，替代以均匀分布；如果为False，则去学习 $P(y=c_k)$。
- class_prior：一个数组。它指定了每个分类的先验概率 $P(y=c_1),P(y=c_2),\cdots,P(y=c_K)$。如果指定了该参数，则每个分类的先验概率不再从数据集中学得。

属性

- class_log_prior_：一个数组对象，形状为(n_classes,)。给出了每个类别调整后的经验概率分布的对数值。

- ❑ feature_log_prob_: 一个数组对象，形状为(n_classes, n_features)。给出了 $P(X^{(j)}/y = c_k)$ 的经验概率分布的对数值。
- ❑ class_count_：一个数组，形状为(n_classes,)，是每个类别包含的训练样本数量。
- ❑ feature_count_：一个数组，形状为(n_classes, n_features)，是训练过程中，每个类别每个特征遇到的样本数。

方法

- ❑ fit(X, y[, sample_weight])：训练模型。
- ❑ partial_fit(X, y[, classes, sample_weight])：追加训练模型。该方法主要用于大规模数据集的训练。此时可以将大数据集划分成若干个小数据集，然后在这些小数据集上连续调用partial_fit方法来训练模型。
- ❑ predict(X)：用模型进行预测，返回预测值。
- ❑ predict_log_proba(X)：返回一个数组，数组的元素依次是X预测为各个类别的概率的对数值。
- ❑ predict_proba(X)：返回一个数组，数组的元素依次是X预测为各个类别的概率值。
- ❑ score(X, y[, sample_weight])：返回在 (X,y)上预测的准确率 (accuracy)。

给出测试伯努利贝叶斯分类器的函数：

```
def test_BernoulliNB(*data):
    X_train,X_test,y_train,y_test=data
    cls=naive_bayes.BernoulliNB()
    cls.fit(X_train,y_train)
    print('Training Score: %.2f' % cls.score(X_train,y_train))
    print('Testing Score: %.2f' % cls.score(X_test, y_test))
```

调用test_BernoulliNB函数，运行结果如下：

```
Training Score: 0.87
Testing Score: 0.85
```

可以看到伯努利贝叶斯分类器对训练数据集的预测准确率为 87%，对测试数据集的预测准确率为 85%。

接着检验不同的 α 对伯努利贝叶斯分类器的预测性能的影响，给出函数：

```
def test_BernoulliNB_alpha(*data):
    X_train,X_test,y_train,y_test=data
    alphas=np.logspace(-2,5,num=200)
    train_scores=[]
    test_scores=[]
    for alpha in alphas:
        cls=naive_bayes.BernoulliNB(alpha=alpha)
        cls.fit(X_train,y_train)
        train_scores.append(cls.score(X_train,y_train))
```

```
        test_scores.append(cls.score(X_test, y_test))

    ## 绘图
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    ax.plot(alphas,train_scores,label="Training Score")
    ax.plot(alphas,test_scores,label="Testing Score")
    ax.set_xlabel(r"$\alpha$")
    ax.set_ylabel("score")
    ax.set_ylim(0,1.0)
    ax.set_title("BernoulliNB")
    ax.set_xscale("log")
    ax.legend(loc="best")
    plt.show()
```

调用test_BernoulliNB_alpha函数。运行结果如图 3.3 所示。为了便于观察，我们将 x 轴设置为对数坐标。可以看到 $\alpha > 100$ 之后，随着 α 的增长，预测准确率在下降。原因与多项式贝叶斯分类器的情况相同。

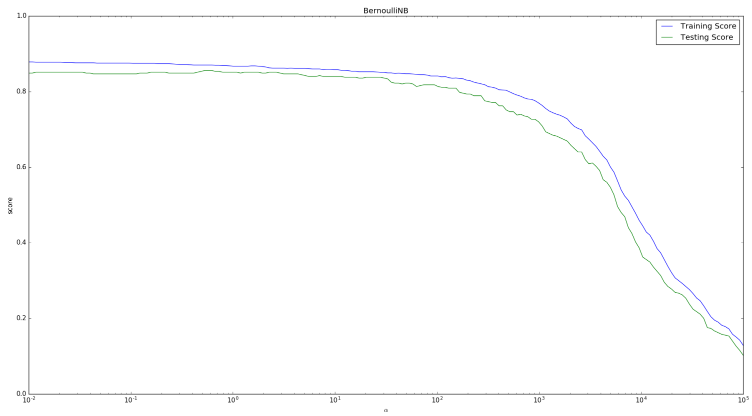


图 3.3 BernoulliNB_alpha

最后考察binarize的参数对伯努利贝叶斯分类器的预测性能的影响。该参数给定了二元化时，0-1的阈值。给出函数：

```
def test_BernoulliNB_binarize(*data):
    X_train,X_test,y_train,y_test=data
    min_x=min(np.min(X_train.ravel()),np.min(X_test.ravel()))-0.1
    max_x=max(np.max(X_train.ravel()),np.max(X_test.ravel()))+0.1
    binarizes=np.linspace(min_x,max_x,endpoint=True,num=100)
    train_scores=[]
    test_scores=[]
    for binarize in binarizes:
        cls=naive_bayes.BernoulliNB(binarize=binarize)
        cls.fit(X_train,y_train)
```

```
        train_scores.append(cls.score(X_train,y_train))
        test_scores.append(cls.score(X_test, y_test))

    ## 绘图
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    ax.plot(binarizes,train_scores,label="Training Score")
    ax.plot(binarizes,test_scores,label="Testing Score")
    ax.set_xlabel("binarize")
    ax.set_ylabel("score")
    ax.set_ylim(0,1.0)
    ax.set_xlim(min_x-1,max_x+1)
    ax.set_title("BernoulliNB")
    ax.legend(loc="best")
    plt.show()
```

指定的binarize的最小值为样本集（包括测试集）所有特征的所有值中的最小值减去0.1，当binarize取最小值时所有特征的所有值都视为 1；指定的binarize的最大值为样本集（包括测试集）所有特征的所有值中的最大值加上 0.1，当binarize取最大值时所有特征的所有值都视为 0。调用test_BernoulliNB_alpha函数。运行结果如图 3.4 所示。可以看到当binarize太小时，预测准确率断崖式下降，这是因为此时所有特征的所有值都视为 0，此时对于伯努利贝叶斯分类器来讲，所有样本的所有特征都是“平坦”的白茫茫一片，样本之前没有任何区分，所以也无从预测。当binarize太大时，预测准确率也是断崖式下降，这是因为此时所有特征的所有值都视为 1，此时对于伯努利贝叶斯分类器来讲样本之前没有任何区分，同样也无从预测。binarize的取值必须在样本集（包括测试集）所有特征的所有值的最小值和最大值之间，且最好能使得二元化之后的特征分布尽可能近似于原始特征的分布。

> 作为一个经验值，可以将binarize 取“(所有特征的所有值的最小值 + 所有特征的所有值的最大值）/ 2”。

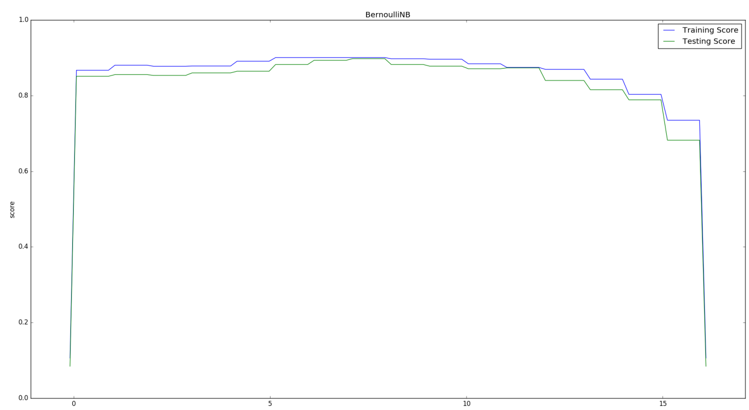


图 3.4　BernoulliNB_binarize

3.3.4 递增式学习 partial_fit 方法

朴素贝叶斯模型可以用来解决大规模的分类问题，其完整的训练集可能不适合放在内存中。为解决这个问题，上述三个分类器都有一个.partial_fit方法，可以动态地增加数据来使用 (即所谓 online classifier)，能够用于递增式学习。

partial_fit的原型为：partial_fit(X, y, classes=None, sample_weight=None)。

- ❑ X：样本数据。
- ❑ y：样本标记。
- ❑ classes：一个数组对象，形状为(n_classes,)，它列出了所有可能的类别。记住：第一次调用pritial_fit时，必须传入该参数。后续的调用不必传入该参数。
- ❑ sample_weight：一个数组对象，形状为(n_samples,)。给出每个样本的权重。如果未指定，则全为 1。

使用该方法时，最好每次数据块都足够大，推荐每次填满整个内存。通过多次连续地调用partial_fit方法，成百上千GB的数据集就可以被切分成一块一块地来进行训练了。

第 4 章

k 近邻法

4.1 概述

k 近邻法（k-Nearest Neighbor，kNN）是机器学习所有算法中理论最简单，最好理解的算法。它是一种基本的分类与回归方法，它的输入为实例的特征向量，通过计算新数据与训练数据特征值之间的距离，然后选取 K（K>=1）个距离最近的邻居进行分类判断（投票法）或者回归。如果 K=1，那么新数据被简单地分配给其近邻的类。

对于分类问题：输出为实例的类别。分类时，对于新的实例，根据其 k 个最近邻的训练实例的类别，通过多数表决等方式进行预测。

对于回归问题：输出为实例的值。回归时，对于新的实例，取其 k 个最近邻的训练实例的平均值为预测值。

k 近邻法分类的直观理解：给定一个训练数据集，对于新的输入实例，在训练集中找到与该实例最邻近的 k 个实例。这 k 个实例的多数属于某个类别，则该输入实例就划分为这个类别。

k 近邻法不具有显式的学习过程，它是直接预测。实际上它是利用训练数据集对特征向量空间进行划分，并且作为其分类的“模型”。

4.2 算法笔记精华

4.2.1 kNN 三要素

k 近邻法的三要素：k 值选择、距离度量和分类决策规则（取均值的决策规则）。

k 值选择

当 $k=1$ 时的 k 近邻算法称为最近邻算法。此时将训练集中与 $\vec{\boldsymbol{x}}$ 最近的点的类别作为 $\vec{\boldsymbol{x}}$ 的分类。

k 值的选择会对 k 近邻法的结果产生重大影响。

- 若 k 值较小，则相当于用较小的邻域中的训练实例进行预测，“学习”的近似误差减小。
  - 优点：只有与输入实例较近的训练实例才会对预测起作用。
  - 缺点：“学习”的估计误差会增大，预测结果会对近邻的实例点非常敏感。若近邻的训练实例点刚好是噪声，则预测会出错。即 k 值的减小意味着模型整体变复杂，易发生过拟合。
- 若 k 值较大，则相当于用较大的邻域中的训练实例进行预测。
  - 优点：减少学习的估计误差。
  - 缺点：学习的近似误差会增大。这时输入实例较远的训练实例也会对预测起作用，使预测发生错误，即 k 值增大意味着模型整体变简单。当 $k=N$ 时，无论输入实例是什么，都将它预测为训练实例中最多的类（即预测结果是一个常量）。此时模型过于简单，完全忽略了训练实例中大量有用的信息。

应用中，k 值一般取一个较小的数值。通常采用交叉验证法来选取最优的 k 值，就是比较不同 k 值时的交叉验证平均误差率，选择误差率最小的那个 k 值。例如选择 $k=1,2,3,\cdots$，对每个 $k=i$ 做若干次交叉验证，计算出平均误差，然后比较、选出最小的那个。

距离度量

kNN 算法要求数据的所有特征都可以做可比较的量化。若在数据特征中存在非数值的类型，必须采取手段将其量化为数值。比如，如果样本特征中包含颜色（红、黑、蓝）一项，颜色之间是没有距离可言的，可通过将颜色转换为灰度值来实现距离计算。另外，样本有多个参数，每一个参数都有自己的定义域和取值范围，它们对距离计算的影响也就不一样，如取值较大的影响力会盖过取值较小的参数。为了公平，样本参数必须做一些归一化处理，最简单的方式就是所有特征的数值都采取归一化处置。

特征空间中两个实例点的距离是两个实例点相似程度的反映。k 近邻模型的特征空间一般是 n 维实数向量空间 $\mathbb{R}^n$。k 近邻模型的特征空间的距离一般为欧氏距离，也可以是一般的 L_p 距离：

$$L_p(\vec{\boldsymbol{x}}_i,\vec{\boldsymbol{x}}_j)=(\sum_{l=1}^{n}|x_i^{(l)}-x_j^{(l)}|^p)^{1/p}$$

$$\vec{\boldsymbol{x}}_i,\vec{\boldsymbol{x}}_j\in\mathcal{X}=\mathbb{R}^n$$

$$\vec{\boldsymbol{x}}_i = (x_i^{(1)}, x_i^{(2)}, \cdots, x_i^{(n)})^T$$
$$\vec{\boldsymbol{x}}_j = (x_j^{(1)}, x_j^{(2)}, \cdots, x_j^{(n)})^T$$
$$p \geqslant 1$$

- 当 $p = 2$ 时，为欧氏距离：$L_2(\vec{\boldsymbol{x}}_i, \vec{\boldsymbol{x}}_j) = (\sum_{l=1}^n |x_i^{(l)} - x_j^{(l)}|^2)^{1/2}$。
- 当 $p = 1$ 时，为曼哈顿距离：$L_1(\vec{\boldsymbol{x}}_i, \vec{\boldsymbol{x}}_j) = \sum_{l=1}^n |x_i^{(l)} - x_j^{(l)}|$。
- 当 $p = \infty$ 时，为各维度距离中的最大值：$L_\infty(\vec{\boldsymbol{x}}_i, \vec{\boldsymbol{x}}_j) = \max_l |x_i^{(l)} - x_j^{(l)}|$。

不同的距离度量所确定的最近邻点是不同的。一般情况下，选欧氏距离作为距离度量，但这只适用于连续变量。在文本分类这种非连续变量情况下，汉明距离可以用来作为度量。通常情况下，如果运用一些特殊的算法来计算度量的话，K 近邻分类的精度可显著提高，如运用大边缘最近邻法或者近邻成分分析法。

分类决策规则

分类决策通常采用多数表决。也可以基于距离的远近进行加权投票，距离越近的样本权重越大。

多数表决规则等价于经验风险最小化。设分类的损失函数为 0 – 1 损失函数，分类函数为 $f: \mathbb{R}^n \to \{c_1, c_2, \cdots, c_K\}$，误分类概率为：$P(Y \neq f(X)) = 1 - P(Y = f(X))$。

给定实例 $\vec{\boldsymbol{x}} \in \mathcal{X}$，其最邻近的 k 个训练点构成集合 $N_k(\vec{\boldsymbol{x}})$。设涵盖 $N_k(\vec{\boldsymbol{x}})$ 区域的类别为 c_j（这是个待求的未知量，但它肯定是 $c_1, c_2, \cdots, c_K$ 之一），则误分类率为：

$$\frac{1}{k} \sum_{\vec{\boldsymbol{x}}_i \in N_k(\vec{\boldsymbol{x}})} I(y_i \neq c_j) = 1 - \frac{1}{k} \sum_{\vec{\boldsymbol{x}}_i \in N_k(\vec{\boldsymbol{x}})} I(y_i = c_j)$$

误分类率就是训练数据的经验风险。要使误分类率最小，即经验风险最小，就要使得 $\sum_{\vec{\boldsymbol{x}}_i \in N_k(\vec{\boldsymbol{x}})} I(y_i = c_j)$ 最大。即多数表决：

$$c_j = \arg\max_{c_j} \sum_{\vec{\boldsymbol{x}}_i \in N_k(\vec{\boldsymbol{x}})} I(y_i = c_j)$$

4.2.2 k 近邻算法

k 近邻法的分类算法描述如下。

- 输入：训练数据集 $T = \{(\vec{\boldsymbol{x}}_1, y_1), (\vec{\boldsymbol{x}}_2, y_2), \cdots, (\vec{\boldsymbol{x}}_N, y_N)\}, \vec{\boldsymbol{x}}_i \in \mathcal{X} \subseteq \mathbb{R}^n$ 为实例的特征向量，$y_i \in \mathcal{Y} = \{c_1, c_2, \cdots, c_K\}$ 为实例的类别，$i = 1, 2, \cdots, N$。给定实例特征向量 $\vec{\boldsymbol{x}}$。
- 输出：实例 $\vec{\boldsymbol{x}}$ 所属的类别 y。

- 步骤
  - 根据给定的距离度量，在 T 中寻找与 $\vec{x}$ 最近邻的 k 个点。定义涵盖这 k 个点的 $\vec{x}$ 的邻域记作 $N_k(\vec{x})$。
  - 从 $N_k(\vec{x})$ 中，根据分类决策规则（如多数表决）决定 $\vec{x}$ 的类别 y：

$$y = \arg\max_{c_j} \sum_{\vec{x}_i \in N_k(\vec{x})} I(y_i = c_j), i = 1, 2, \cdots, N; j = 1, 2, \cdots, K$$

  其中 I 为指示函数：$I(true) = 1, I(false) = 0$。上式中，对于 $y_i, i = 1, 2, \cdots, N$ 只有 $\vec{x}_i \in N_k(\vec{x})$ 中的样本点才考虑。

k 近邻法的学习有一个明显的特点：它没有显式的训练过程。它在训练阶段仅仅将样本保存起来，训练时间开销为零，等到收到测试样本后再进行处理。

4.2.3 kd 树

k 近邻法中如何对训练数据进行快速 k 近邻搜索是个问题。最简单粗暴的方法是：线性扫描。通过计算输入样本与每个训练样本的距离，来找出最近邻的 k 个训练样本。当训练集很大时，计算非常耗时。常用的解决方法是使用 kd 树，它可以大幅提高 k 近邻搜索的效率。

kd 树是二叉树，表示对 k 维空间的一个划分。构造平衡 kd 树的算如下。

- 输入：k 维空间数据集 $T = \{\vec{x}_1, \vec{x}_2, \cdots, \vec{x}_N\}, \vec{x}_i \in \mathcal{X} \subseteq \mathbb{R}^k$。
- 输出：kd 树。
- 算法步骤
  - 开始：构造根节点。选择 $x^{(1)}$ 为轴，以 T 中所有样本的 $x^{(1)}$ 坐标的中位数 $x^{(1)*}$ 作为切分点，将根节点的超矩形切分为两个子区域（切分超平面 $x^{(1)} = x^{(1)*}$）。本次切分产生深度为 1 的左、右子节点。左子节点对应于坐标 $x^{(1)} < x^{(1)*}$ 的子区域；右子节点对应于坐标 $x^{(1)} > x^{(1)*}$ 的子区域；落在切分超平面上的点（$x^{(1)} = x^{(1)*}$）保存在根节点。
  - 重复：对深度为 j 的子节点，选择 $x^{(l)}$ 为切分的坐标轴，$l = j \pmod k + 1$。本次切分之后，树的深度为 $j + 1$ 。这里取模，而不是 $l = j + 1$ ，是因为树的深度可以超过维度 k ，此时切分轴又重复回到 $x^{(1)}$，轮转坐标轴进行切分。
  - 结束：直到所有节点的两个子域中没有样本存在时，切分停止。此时得到 kd 树。

使用 kd 树的算法相对复杂。用 kd 树的最近邻搜索算法（k 近邻搜索依次类推）如下。

- 输入
  - kd 树。
  - 样本 $\vec{x}$。
- 输出：样本 $\vec{x}$ 的最近邻点。

- 步骤
  - 在 kd 树中找到包含测试点 $\vec{x}$ 的叶节点。方法是：从根节点出发，递归向下访问 kd 树：
    - 若测试点 $\vec{x}$ 当前维度的坐标小于切分点的坐标，则查找当前节点的左子节点；
    - 若测试点 $\vec{x}$ 当前维度的坐标大于切分点的坐标，则查找当前节点的右子节点，在访问过程中记录下访问的各结点的顺序（以便于后面的回退）。
  - 以此叶节点为“当前最近” $\vec{x}_{nst}$。真实最近点一定在 $\vec{x}$ 与“当前最近点”构成的超球体内。$\vec{x}$ 为球心。
  - 设当前考察的节点为 $\vec{x}_i$，递归向上回退，设回退弹出的节点为 $\vec{x}_{inew}$（每次回退都是退到 kd 树的父节点），考察结点 $\vec{x}_{inew}$ 所在的超平面与以 $\vec{x}$ 为球心、以 $\vec{x}$ 到当前最近点 $\vec{x}_{nst}$ 的距离为半径的超球体是否相交：
    - 若相交
      - ★ 若 $\vec{x}_i$ 是 $\vec{x}_{inew}$ 的左子节点，则进入 $\vec{x}_{inew}$ 的右子节点，然后先进行向下搜索，再然后向上回退。
      - ★ 若 $\vec{x}_i$ 是 $\vec{x}_{inew}$ 的右子节点，则进入 $\vec{x}_{inew}$ 的左子节点，然后先进行向下搜索，再然后向上回退。
    - 若不相交，则直接回退。
  - 当回退到根节点时，搜索结束。最后的“当前最近点”即为 $\vec{x}$ 的最近邻点。

kd 树搜索的平均计算复杂度为 $O(\log N)$，N 为训练集大小。kd 树适合 $N \gg k$ 的情形。当 N 与维度 k 接近时，搜索的效率接近线性扫描。

4.3　Python 实践

首先导入包：

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import neighbors, datasets,cross_validation
```

然后给出加载数据集的函数：

```
def load_classification_data():
    digits=datasets.load_digits()
    X_train=digits.data
    y_train=digits.target
    return cross_validation.train_test_split(X_train, y_train,test_size=0.25,
            random_state=0,stratify=y_train)

def create_regression_data(n):
    X =5 * np.random.rand(n, 1)
    y = np.sin(X).ravel()
```

```
    y[::5] += 1 * (0.5 - np.random.rand(int(n/5)))
    return cross_validation.train_test_split(X, y,test_size=0.25,random_state=0)
```

其中，load_classification_data函数使用的是scikit-learn自带的手写识别数据集Digit Dataset。该数据集由 1797 张样本图片组成。每张样本图片都是一个 8×8 大小的手写数字位图。create_regression_data函数是在sin(X)基础上添加噪声生成的。

kNN 分类 KNeighborsClassifier

scikit-learn中提供了一个KNeighborsClassifier类来实现 k 近邻法分类模型，其原型为：

```
sklearn.neighbors.KNeighborsClassifier(n_neighbors=5, weights='uniform',
        algorithm='auto', leaf_size=30, p=2, metric='minkowski',
        metric_params=None, n_jobs=1, **kwargs)
```

参数

- n_neighbors：一个整数，指定 k 值。
- weights：一字符串或者可调用对象，指定投票权重类型。即这些邻居投票权可以为相同或者不同。
  - 'uniform'：本节点的所有邻居节点的投票权重都相等。
  - 'distance'：本节点的所有邻居节点的投票权重与距离成反比。即越近的节点，其投票权重越大。
  - [callable]：一个可调用对象。它传入距离的数组，返回同样形状的权重数组。
- algorithm：一字符串，指定计算最近邻的算法，可以为如下。
  - 'ball_tree'：使用 BallTree算法。
  - 'kd_tree：使用 KDTree算法。
  - 'brute'：使用暴力搜索法。
  - 'auto'：自动决定最合适的算法。
- leaf_size：一个整数，指定BallTree或者 KDTree叶节点规模。它影响树的构建和查询速度。
- metric：一个字符串，指定距离度量。默认为'minkowski'距离。
- p: 整数值，指定在'Minkowski'度量上的指数。如果 p=1，对应曼哈顿距离；如果'p=2'，对应欧拉距离。
- n_jobs：并行性。默认为 −1 表示派发任务到所有计算机的 CPU 上。

方法

- fit(X,y)：训练模型。
- predict(X)：使用模型来预测，返回待预测样本的标记。
- score(X,y)：返回在 (X,y)上预测的准确率 (accuracy)。

- predict_proba(X)：返回样本为每种标记的概率。
- kneighbors([X, n_neighbors, return_distance])：返回样本点的 k 近邻点。如果 return_distance=True，同时还返回到这些近邻点的距离。
- kneighbors_graph([X, n_neighbors, mode])：返回样本点的连接图。

首先使用KNeighborsClassifier，给出测试函数：

```
def test_KNeighborsClassifier(*data):
    X_train,X_test,y_train,y_test=data
    clf=neighbors.KNeighborsClassifier()
    clf.fit(X_train,y_train)
    print("Training Score:%f"%clf.score(X_train,y_train))
    print("Testing Score:%f"%clf.score(X_test,y_test))
```

然后调用test_KNeighborsClassifier函数：

```
X_train,X_test,y_train,y_test=load_classification_data()
test_KNeighborsClassifier(X_train,X_test,y_train,y_test)
```

结果如下：

```
Training Score:0.991085
Testing Score:0.980044
```

可以看到 k近邻法对于测试集的数据预测准确率高达 98.0044%，对于训练集的拟合准确率高达 99.1085%。

然后考察 k 值以及投票策略对于预测性能的影响，给出测试函数：

```
def test_KNeighborsClassifier_k_w(*data):
    X_train,X_test,y_train,y_test=data
    Ks=np.linspace(1,y_train.size,num=100,endpoint=False,dtype='int')
    weights=['uniform','distance']

    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    for weight in weights:
        training_scores=[]
        testing_scores=[]
        for K in Ks:
            clf=neighbors.KNeighborsClassifier(weights=weight,n_neighbors=K)
            clf.fit(X_train,y_train)
            testing_scores.append(clf.score(X_test,y_test))
            training_scores.append(clf.score(X_train,y_train))
        ax.plot(Ks,testing_scores,label="testing score:weight=%s"%weight)
        ax.plot(Ks,training_scores,label="training score:weight=%s"%weight)
    ax.legend(loc='best')
    ax.set_xlabel("K")
```

```
    ax.set_ylabel("score")
    ax.set_ylim(0,1.05)
    ax.set_title("KNeighborsClassifier")
    plt.show()
```

同样地调用test_KNeighborsClassifier_k_w函数，结果如图 4.1 所示。

可以看到使用uniform投票策略的情况下（即投票权重都相同），分类器随着 k 的增长，预测性能稳定下降。这是因为当 k 增大时，输入实例较远的训练实例也会对预测起作用，使预测发生错误。

在使用distance投票策略的情况下（即投票权重与距离成反比），分类器随着 k 的增长，对测试集的预测性能相对比较稳定。这是因为虽然 k 增大时，输入实例较远的训练实例也会对预测起作用，但因为距离较远，其影响小得多（权重很小）。

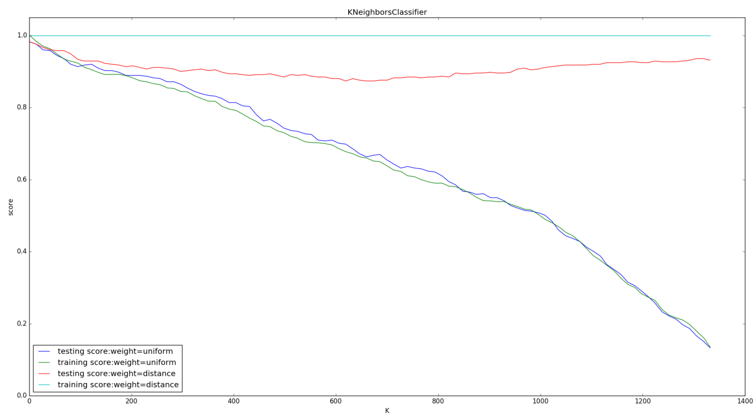


图 4.1 KNeighborsClassifier_k_w

然后考察 p 值（即距离函数的形式）对于预测性能的影响，给出测试函数：

```
def test_KNeighborsClassifier_k_p(*data):
    X_train,X_test,y_train,y_test=data
    Ks=np.linspace(1,y_train.size,endpoint=False,dtype='int')
    Ps=[1,2,10]

    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    for P in Ps:
        training_scores=[]
        testing_scores=[]
        for K in Ks:
            clf=neighbors.KNeighborsClassifier(p=P,n_neighbors=K)
            clf.fit(X_train,y_train)
            testing_scores.append(clf.score(X_test,y_test))
            training_scores.append(clf.score(X_train,y_train))
        ax.plot(Ks,testing_scores,label="testing score:p=%d"%P)
```

```
        ax.plot(Ks,training_scores,label="training score:p=%d"%P)
    ax.legend(loc='best')
    ax.set_xlabel("K")
    ax.set_ylabel("score")
    ax.set_ylim(0,1.05)
    ax.set_title("KNeighborsClassifier")
    plt.show()
```

同样地调用test_KNeighborsClassifier_k_p函数，结果如图 4.2 所示。可以看到 p 参数对于分类器的预测性能没有任何影响。因为 $L_p(\vec{x}_i,\vec{x}_j) = (\sum_{l=1}^{n} |x_i^{(l)} - x_j^{(l)}|^p)^{1/p}$，当 $p = 1$ 时如果 $\vec{x}_j$ 是 $\vec{x}_i$ 的最近的点；则当 p 为其他值时，该结论也成立。

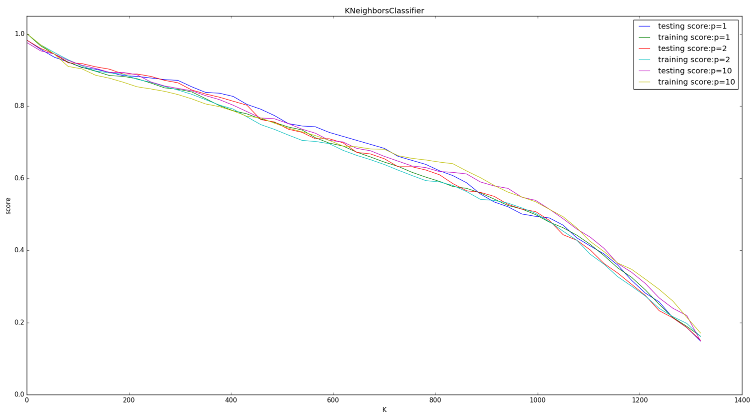


图 4.2　KNeighborsClassifier_k_p

kNN 回归 KNeighborsRegressor

scikit-learn中提供了一个KNeighborsRegressor类来实现 k 近邻法回归模型，其原型为：

```
sklearn.neighbors.KNeighborsRegressor(n_neighbors=5, weights='uniform',
        algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None,
        n_jobs=1, **kwargs)
```

参数

- n_neighbors：一个整数，指定 k 值。
- weights：一字符串或者可调用对象，指定投票权重类型。即这些邻居投票权可以为相同或者不同。
  - 'uniform'：本节点所有邻居节点的投票权重都相等。
  - 'distance'：本节点的所有邻居节点的投票权重与距离成反比。即越近的节点，其投票权重越大。
  - [callable]：一个可调用对象。它传入距离的数组，返回同样形状的权重数组。

- algorithm：一字符串，指定计算最近邻的算法，可以为如下。
  - 'ball_tree'：使用 BallTree算法。
  - 'kd_tree：使用 KDTree算法。
  - 'brute'：使用暴力搜索法。
  - 'auto'：自动决定最合适的算法。
- leaf_size：一个整数，指定BallTree或者 KDTree叶节点规模。它影响树的构建和查询速度。
- metric：一个字符串，指定距离度量。默认为'minkowski'距离。
- p: 一个整数值，指定在'Minkowski'度量上的指数。如果 p=1，对应曼哈顿距离；如果 'p=2'，对应欧拉距离。
- n_jobs：一个整数，指定并行性。默认为 -1 ，表示派发任务到所有计算机的 CPU 上。

方法

- fit(X,y)：训练模型。
- predict(X)：使用模型来预测，返回待预测样本的标记。
- score(X,y)：返回预测性能得分。设预测集为 T_{test}，真实值为 y_i，真实值的均值为 $\bar{y}$，预测值为 $\hat{y}_i$，则：

$$score = 1 - \frac{\sum_{T_{test}}(y_i - \hat{y}_i)^2}{(y_i - \bar{y})^2}$$

  - score不超过 1，但是可能为负值（预测效果太差）。
  - score越大，预测性能越好。
- kneighbors([X, n_neighbors, return_distance])：返回样本点的 k 近邻点。如果 return_distance=True，同时还返回到这些近邻点的距离。
- kneighbors_graph([X, n_neighbors, mode])：返回样本点的连接图。

其参数意义以及实例方法与KNeighborsClassifier几乎完全相同。两者区别在于回归分析和分类决策的不同：

- KNeighborsClassifier将待预测样本点最近邻的 k 个训练样本点中出现次数最多的分类作为待预测样本点的分类。
- KNeighborsRegressor将待预测样本点最近邻的 k 个训练样本点的平均值作为待预测样本点的值。

首先使用KNeighborsRegressor，给出测试函数：

```
def test_KNeighborsRegressor(*data):
    X_train,X_test,y_train,y_test=data
    regr=neighbors.KNeighborsRegressor()
    regr.fit(X_train,y_train)
    print("Training Score:%f"%regr.score(X_train,y_train))
    print("Testing Score:%f"%regr.score(X_test,y_test))
```

然后我们调用test_KNeighborsRegressor函数：

```
X_train,X_test,y_train,y_test=create_regression_data(1000)
test_KNeighborsRegressor(X_train,X_test,y_train,y_test)
```

这里我们生成了 1000 个样本数据。结果如下：

```
Training Score:0.974401
Testing Score:0.956201
```

可以看到回归器对于测试集的数据预测得分为 0.956201，对于训练集的预测得分为 0.974401。

然后考察 k 值以及投票策略对预测性能的影响，给出测试函数：

```
def test_KNeighborsRegressor_k_w(*data):
    X_train,X_test,y_train,y_test=data
    Ks=np.linspace(1,y_train.size,num=100,endpoint=False,dtype='int')
    weights=['uniform','distance']

    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    for weight in weights:
        training_scores=[]
        testing_scores=[]
        for K in Ks:
            regr=neighbors.KNeighborsRegressor(weights=weight,n_neighbors=K)
            regr.fit(X_train,y_train)
            testing_scores.append(regr.score(X_test,y_test))
            training_scores.append(regr.score(X_train,y_train))
        ax.plot(Ks,testing_scores,label="testing score:weight=%s"%weight)
        ax.plot(Ks,training_scores,label="training score:weight=%s"%weight)
    ax.legend(loc='best')
    ax.set_xlabel("K")
    ax.set_ylabel("score")
    ax.set_ylim(0,1.05)
    ax.set_title("KNeighborsRegressor")
    plt.show()
```

同样地调用test_KNeighborsRegressor_k_w函数，结果如图 4.3 所示。其讨论与 KNeighborsClassifier 相同。

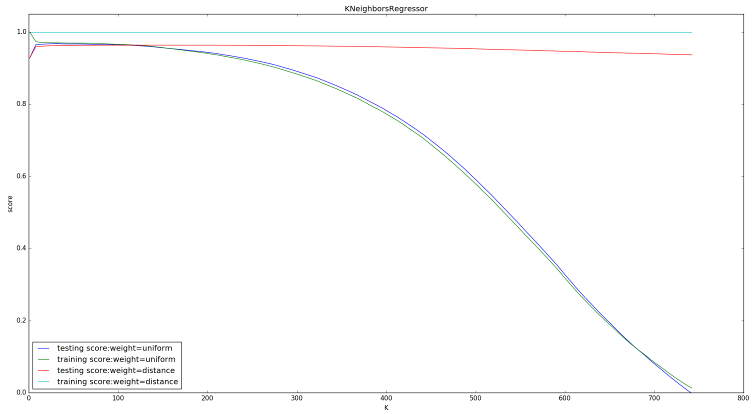


图 4.3 KNeighborsRegressor_k_w

然后考察 p 值（即距离函数的形式）对于预测性能的影响，给出测试函数：

```
def test_KNeighborsRegressor_k_p(*data):
    X_train,X_test,y_train,y_test=data
    Ks=np.linspace(1,y_train.size,endpoint=False,dtype='int')
    Ps=[1,2,10]

    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    for P in Ps:
        training_scores=[]
        testing_scores=[]
        for K in Ks:
            regr=neighbors.KNeighborsRegressor(p=P,n_neighbors=K)
            regr.fit(X_train,y_train)
            testing_scores.append(regr.score(X_test,y_test))
            training_scores.append(regr.score(X_train,y_train))
        ax.plot(Ks,testing_scores,label="testing score:p=%d"%P)
        ax.plot(Ks,training_scores,label="training score:p=%d"%P)
    ax.legend(loc='best')
    ax.set_xlabel("K")
    ax.set_ylabel("score")
    ax.set_ylim(0,1.05)
    ax.set_title("KNeighborsRegressor")
    plt.show()
```

同样地调用test_KNeighborsRegressor_k_p函数，结果如图 4.4 所示，其讨论与 KNeighborsClassifier 相同。

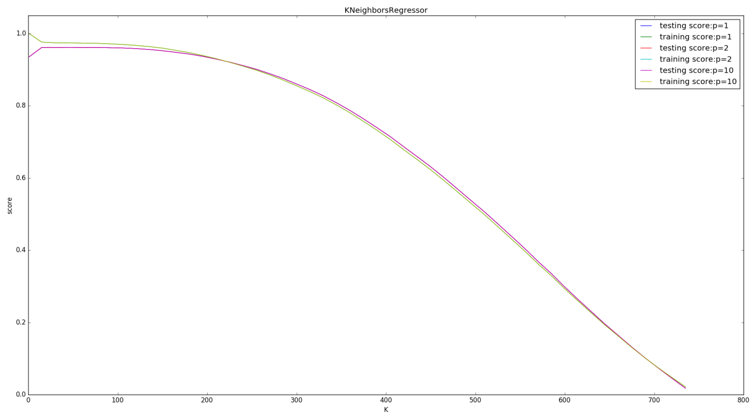


图 4.4 KNeighborsRegressor_k_p

第 5 章

数据降维

5.1 概述

数据降维是机器学习领域中非常重要的内容。所谓的降维就是指采用某种映射方法，将原高维空间中的数据点映射到低维度的空间中。降维的本质是学习一个映射函数 $f: x \to y$，其中 x 是原始数据点的表达，目前多使用向量表达形式。y 是数据点映射后的低维向量表达，通常 y 的维度小于 x 的维度。映射函数 f 可能是显式的或隐式的、线性的或非线性的。

目前大部分降维算法处理向量表达的数据，也有一些降维算法处理高阶张量表达的数据。之所以使用降维后的数据表示是因为在原始的高维空间中，包含冗余信息以及噪声信息，在实际应用例如图像识别中造成了误差，降低了准确率；而通过降维，我们希望减少冗余信息所造成的误差，提高识别（或其他应用）的精度。又或者希望通过降维算法来寻找数据内部的本质结构特征。

在很多算法中，降维算法成为了数据预处理的一部分，如下面要讲到的 PCA 算法。事实上，有一些算法如果没有降维预处理，其实是很难得到很好效果的。

5.2 算法笔记精华

5.2.1 维度灾难与降维

对于 k 近邻法，最好要求样本点比较密集。理论上给定测试样本 $\vec{\boldsymbol{x}}$，我们希望在 $\vec{\boldsymbol{x}}$ 附近很近的距离 $\delta > 0$ 范围内总能找到一个训练样本 $\vec{\boldsymbol{z}}$。假设 $\delta = 0.001$，且所有特征的取值范围都是 $[0,1]$：

- 若样本只有一个特征，则需要 1000 个均匀分布的训练样本。此时任何测试样本在其附近 δ 距离范围内总能找到一个训练样本；
- 若样本只有 10 个特征，则需要 10^{30} 个均匀分布的训练样本。此时任何测试样本在其附近 δ 距离范围内总能找到一个训练样本。

如果特征维度成千上万，则需要的训练样本的数量几乎不可能满足。而且高维空间的距离计算也比较麻烦。在高维情形下出现的数据样本稀疏、距离计算困难等问题是所有机器学习方法共同面临的严重障碍，称为“维度灾难”(curse of dimensionality)。可以通过降维 (dimension reduction) 来缓解这个问题。

前面章节介绍的线性判别分析（Linear Discriminant Analysis，LDA）是一种经典的监督降维算法。主成分分析PCA是一种经典的无监督降维算法。

对于降维效果的评估，通常是比较降维前后学习器的性能。如果性能有所提高，则认为降维起了作用。如果将维数降至二维或者三维，则可以通过可视化技术来直观地判断降维的效果。

5.2.2 主成分分析（PCA）

PCA 原理

主成分分析（Principal Component Analysis，PCA）是最常用的一种降维方法。为了便于维度变换，有如下假设。

- 假设样本数据是 n 维的。
- 假设原始坐标系为：由标准正交基向量 $\{\vec{\boldsymbol{i}}_1,\vec{\boldsymbol{i}}_2,\cdots,\vec{\boldsymbol{i}}_n\}$ 张成的空间，其中 $||\vec{\boldsymbol{i}}_s||=1;\vec{\boldsymbol{i}}_s\cdot\vec{\boldsymbol{i}}_t=0,s\neq t$。
- 假设经过线性变换后的新坐标系为：由标准正交基向量 $\{\vec{\boldsymbol{j}}_1,\vec{\boldsymbol{j}}_2,\cdots,\vec{\boldsymbol{j}}_n\}$ 张成的空间，其中 $||\vec{\boldsymbol{j}}_s||=1;\vec{\boldsymbol{j}}_s\cdot\vec{\boldsymbol{j}}_t=0,s\neq t$。

根据定义，有：

$$\vec{\boldsymbol{j}}_s=(\vec{\boldsymbol{i}}_1,\vec{\boldsymbol{i}}_2,\cdots,\vec{\boldsymbol{i}}_n)\begin{bmatrix}\vec{\boldsymbol{j}}_s\cdot\vec{\boldsymbol{i}}_1\\ \vdots\\ \vec{\boldsymbol{j}}_s\cdot\vec{\boldsymbol{i}}_n\end{bmatrix},s=1,2,\cdots,n$$

记 $\boldsymbol{w}_s=(\vec{\boldsymbol{j}}_s\cdot\vec{\boldsymbol{i}}_1,\cdots,\vec{\boldsymbol{j}}_s\cdot\vec{\boldsymbol{i}}_n)^T$（它是一个列向量，但是为了与基向量做区分，这里没有给出向量的箭头符号），其各分量就是基向量 $\vec{\boldsymbol{j}}_s$ 在原始坐标系 $\{\vec{\boldsymbol{i}}_1,\vec{\boldsymbol{i}}_2,\cdots,\vec{\boldsymbol{i}}_n\}$ 中的投影。即：$\vec{\boldsymbol{j}}_s=(\vec{\boldsymbol{i}}_1,\vec{\boldsymbol{i}}_2,\cdots,\vec{\boldsymbol{i}}_n)\boldsymbol{w}_s$。根据标准正交基的性质，有：

- $||\boldsymbol{w}_s||=1,s=1,2,\cdots,n;$
- $\boldsymbol{w}_s\cdot\boldsymbol{w}_t=0,s\neq t$

根据定义有：$(\vec{\boldsymbol{j}}_1,\vec{\boldsymbol{j}}_2,\cdots,\vec{\boldsymbol{j}}_n)=(\vec{\boldsymbol{i}}_1,\vec{\boldsymbol{i}}_2,\cdots,\vec{\boldsymbol{i}}_n)(\boldsymbol{w}_1,\boldsymbol{w}_2,\cdots,\boldsymbol{w}_n)$。令坐标变换矩阵 $\boldsymbol{W}$ 为：

$$\boldsymbol{W}=(\boldsymbol{w}_1,\boldsymbol{w}_2,\cdots,\boldsymbol{w}_n)=\begin{bmatrix}\vec{\boldsymbol{j}}_1\cdot\vec{\boldsymbol{i}}_1 & \vec{\boldsymbol{j}}_2\cdot\vec{\boldsymbol{i}}_1 & \cdots & \vec{\boldsymbol{j}}_n\cdot\vec{\boldsymbol{i}}_1\\ \vec{\boldsymbol{j}}_1\cdot\vec{\boldsymbol{i}}_2 & \vec{\boldsymbol{j}}_2\cdot\vec{\boldsymbol{i}}_2 & \cdots & \vec{\boldsymbol{j}}_n\cdot\vec{\boldsymbol{i}}_2\\ \vdots & \vdots & \ddots & \vdots\\ \vec{\boldsymbol{j}}_1\cdot\vec{\boldsymbol{i}}_n & \vec{\boldsymbol{j}}_2\cdot\vec{\boldsymbol{i}}_n & \cdots & \vec{\boldsymbol{j}}_n\cdot\vec{\boldsymbol{i}}_n\end{bmatrix}$$

则有：$(\vec{\boldsymbol{j}}_1,\vec{\boldsymbol{j}}_2,\cdots,\vec{\boldsymbol{j}}_n)=(\vec{\boldsymbol{i}}_1,\vec{\boldsymbol{i}}_2,\cdots,\vec{\boldsymbol{i}}_n)\boldsymbol{W}$。$\boldsymbol{W}$ 的第 s 列就是 $\vec{\boldsymbol{j}}_s$ 在原始坐标系 $\{\vec{\boldsymbol{i}}_1,\vec{\boldsymbol{i}}_2,\cdots,\vec{\boldsymbol{i}}_n\}$ 中的投影。且有 $\boldsymbol{W}=\boldsymbol{W}^T,\boldsymbol{W}\boldsymbol{W}^T=\boldsymbol{I}$（即它的逆矩阵就是它的转置）。

假设样本点 $\vec{\boldsymbol{x}}_i$ 在原始坐标系中的表示为：

$$\vec{\boldsymbol{x}}_i=(\vec{\boldsymbol{i}}_1,\vec{\boldsymbol{i}}_2,\cdots,\vec{\boldsymbol{i}}_n)\begin{bmatrix}x_i^{(1)}\\ x_i^{(2)}\\ \vdots\\ x_i^{(n)}\end{bmatrix}$$

令 $\boldsymbol{x}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(n)})^T$，则 $\vec{\boldsymbol{x}}_i=(\vec{\boldsymbol{i}}_1,\vec{\boldsymbol{i}}_2,\cdots,\vec{\boldsymbol{i}}_n)\boldsymbol{x}_i$。

假设样本点 $\vec{\boldsymbol{x}}_i$ 在新坐标系中的表示为：

$$\vec{\boldsymbol{x}}_i=(\vec{\boldsymbol{j}}_1,\vec{\boldsymbol{j}}_2,\cdots,\vec{\boldsymbol{j}}_n)\begin{bmatrix}z_i^{(1)}\\ z_i^{(2)}\\ \vdots\\ z_i^{(n)}\end{bmatrix}$$

令 $\boldsymbol{z}_i=(z_i^{(1)},z_i^{(2)},\cdots,z_i^{(n)})^T$，则 $\vec{\boldsymbol{x}}_i=(\vec{\boldsymbol{j}}_1,\vec{\boldsymbol{j}}_2,\cdots,\vec{\boldsymbol{j}}_n)\boldsymbol{z}_i$。根据 $\vec{\boldsymbol{x}}_i=\vec{\boldsymbol{x}}_i$，我们有：

$$(\vec{\boldsymbol{j}}_1,\vec{\boldsymbol{j}}_2,\cdots,\vec{\boldsymbol{j}}_n)\boldsymbol{z}_i=(\vec{\boldsymbol{i}}_1,\vec{\boldsymbol{i}}_2,\cdots,\vec{\boldsymbol{i}}_n)\boldsymbol{W}\boldsymbol{z}_i=(\vec{\boldsymbol{i}}_1,\vec{\boldsymbol{i}}_2,\cdots,\vec{\boldsymbol{i}}_n)\boldsymbol{x}_i$$

于是有：$\boldsymbol{z}_i=\boldsymbol{W}^{-1}\boldsymbol{x}_i=\boldsymbol{W}^T\boldsymbol{x}_i$。则有：

$$z_i^{(s)}=\boldsymbol{w}_s^T\boldsymbol{x}_i$$

丢弃其中的部分坐标，将维度降低到 $d<n$，则样本点 $\vec{\boldsymbol{x}}_i$ 在低维坐标系中的坐标为 $\boldsymbol{z}_i'=(z_i^{(1)},z_i^{(2)},\cdots,z_i^{(d)})^T$。现在的问题是：最好丢弃哪些坐标？我们的想法是：基于降维之后的坐标重构样本时，尽量要与原始样本相近。

> 这里始终考虑丢弃最末尾的维度。假设随机挑选要丢弃的维度，则总可以进行线性变换，使得这些被丢弃的维度位于最末尾。

若基于降维后的坐标 $\boldsymbol{z}_i'$ 来重构 $\vec{\boldsymbol{x}}_i$：

$$\hat{\vec{\boldsymbol{x}}}_i = (\vec{\boldsymbol{j}}_1, \vec{\boldsymbol{j}}_2, \cdots, \vec{\boldsymbol{j}}_d)\begin{bmatrix} z_i^{(1)} \\ z_i^{(2)} \\ \vdots \\ z_i^{(d)} \end{bmatrix} = (\vec{\boldsymbol{i}}_1, \vec{\boldsymbol{i}}_2, \cdots, \vec{\boldsymbol{i}}_n)(\boldsymbol{w}_1, \boldsymbol{w}_2, \cdots, \boldsymbol{w}_d)\begin{bmatrix} z_i^{(1)} \\ z_i^{(2)} \\ \vdots \\ z_i^{(d)} \end{bmatrix}$$

$$= (\vec{\boldsymbol{i}}_1, \vec{\boldsymbol{i}}_2, \cdots, \vec{\boldsymbol{i}}_n)(\boldsymbol{w}_1, \boldsymbol{w}_2, \cdots, \boldsymbol{w}_d)\begin{bmatrix} \boldsymbol{w}_1^T \cdot \boldsymbol{x}_i \\ \boldsymbol{w}_2^T \cdot \boldsymbol{x}_i \\ \vdots \\ \boldsymbol{w}_d^T \cdot \boldsymbol{x}_i \end{bmatrix}$$

$$= (\vec{\boldsymbol{i}}_1, \vec{\boldsymbol{i}}_2, \cdots, \vec{\boldsymbol{i}}_n)(\boldsymbol{w}_1, \boldsymbol{w}_2, \cdots, \boldsymbol{w}_d)\begin{bmatrix} \boldsymbol{w}_1^T \\ \boldsymbol{w}_2^T \\ \vdots \\ \boldsymbol{w}_d^T \end{bmatrix} \cdot \boldsymbol{x}_i$$

令 $\boldsymbol{W}_d = (\boldsymbol{w}_1, \boldsymbol{w}_2, \cdots, \boldsymbol{w}_d)$，即它是坐标变换矩阵 $\boldsymbol{W}$ 的前 d 列，则：

$$\hat{\vec{\boldsymbol{x}}}_i = (\vec{\boldsymbol{i}}_1, \vec{\boldsymbol{i}}_2, \cdots, \vec{\boldsymbol{i}}_n)\boldsymbol{W}_d \boldsymbol{W}_d^T \boldsymbol{x}_i$$

考虑整个训练集，原样本点 $\vec{\boldsymbol{x}}_i$ 和基于投影重构的样本点 $\hat{\vec{\boldsymbol{x}}}_i$ 之间的距离为（即所有重构的样本点与原样本点的整体误差）：

$$\sum_{i=1}^{N} ||\hat{\vec{\boldsymbol{x}}}_i - \vec{\boldsymbol{x}}_i||_2^2 = \sum_{i=1}^{N} ||\boldsymbol{x}_i - \boldsymbol{W}_d \boldsymbol{W}_d^T \boldsymbol{x}_i||_2^2$$

考虑：

$$\boldsymbol{W}_d \boldsymbol{W}_d^T \boldsymbol{x}_i = (\boldsymbol{w}_1, \boldsymbol{w}_2, \cdots, \boldsymbol{w}_d)\begin{bmatrix} \boldsymbol{w}_1^T \\ \boldsymbol{w}_2^T \\ \vdots \\ \boldsymbol{w}_d^T \end{bmatrix} \boldsymbol{x}_i = \sum_{s=1}^{d} \boldsymbol{w}_s(\boldsymbol{w}_s^T \boldsymbol{x}_i)$$

由于 $\boldsymbol{w}_s^T \boldsymbol{x}_i$ 是标量，所以有：

$$\boldsymbol{W}_d \boldsymbol{W}_d^T \boldsymbol{x}_i = \sum_{s=1}^{d} (\boldsymbol{w}_s^T \boldsymbol{x}_i)\boldsymbol{w}_s$$

由于 $\boldsymbol{w}_s^T \boldsymbol{x}_i$ 是标量，所以它的转置等于它本身，所以有：

$$\boldsymbol{W}_d \boldsymbol{W}_d^T \boldsymbol{x}_i = \sum_{s=1}^{d} (\boldsymbol{x}_i^T \boldsymbol{w}_s)\boldsymbol{w}_s$$

于是有：

$$\sum_{i=1}^{N}||\hat{\vec{x}}_i-\vec{x}_i||_2^2=\sum_{i=1}^{N}||\boldsymbol{x}_i-\boldsymbol{W}_d\boldsymbol{W}_d^T\boldsymbol{x}_i||_2^2=\sum_{i=1}^{N}||\boldsymbol{x}_i-\sum_{s=1}^{d}(\boldsymbol{x}_i^T\boldsymbol{w}_s)\boldsymbol{w}_s||_2^2$$

定义矩阵 $\boldsymbol{X}=(\boldsymbol{x}_1,\boldsymbol{x}_2,\cdots,\boldsymbol{x}_N)$，即矩阵 $\boldsymbol{X}$ 的第 i 列就是 $\boldsymbol{x}_i$。则可以证明：

$$||\boldsymbol{X}^T-\boldsymbol{X}^T\boldsymbol{W}_d\boldsymbol{W}_d^T||_F^2=\sum_{i=1}^{N}||\boldsymbol{x}_i-\sum_{s=1}^{d}(\boldsymbol{x}_i^T\boldsymbol{w}_s)\boldsymbol{w}_s||_2^2$$

其中，$||\cdot||_F$ 为矩阵的Frobenius范数。接下来的证明过程中，要用到矩阵的 F范数和矩阵的迹的性质：

- 矩阵 $\boldsymbol{A}$ 的 F范数定义为：$||\boldsymbol{A}||_F=\sqrt{\sum_i\sum_j a_{ij}^2}$，即矩阵所有元素的平方和的开方。F范数的性质有：
  - $||\boldsymbol{A}||_F=||\boldsymbol{A}^T||_F$。
  - $||\boldsymbol{A}||_F=tr(A^TA)$，tr为矩阵的迹。
- 对于方阵，矩阵的迹定义为：$tr(A)=\sum_i a_{ii}$，即矩阵对角线元素之和。矩阵的迹的性质有：
  - $tr(\boldsymbol{A})=tr(\boldsymbol{A}^T)$。
  - $tr(\boldsymbol{A}\pm\boldsymbol{B})=tr(\boldsymbol{A})\pm tr(\boldsymbol{B})$。
  - 若 $\boldsymbol{A}$ 为 $m\times n$ 阶矩阵，$\boldsymbol{B}$ 为 $n\times m$ 阶矩阵，则 $tr(\boldsymbol{AB})=tr(\boldsymbol{BA})$。
  - 矩阵的迹等于矩阵的特征值之和：$tr(\boldsymbol{A})=\lambda_1+\cdots+\lambda_n$。
  - 对任何正整数 k 有：$tr(\boldsymbol{A}^k)=\lambda_1^k+\cdots+\lambda_n^k$。

证明过程如下：

$$\begin{aligned}&\boldsymbol{X}^T-\boldsymbol{X}^T\boldsymbol{W}_d\boldsymbol{W}_d^T\\
&=\begin{bmatrix}x_1^{(1)}&x_1^{(2)}&\cdots&x_1^{(n)}\\x_2^{(1)}&x_2^{(2)}&\cdots&x_2^{(n)}\\\vdots&\vdots&\ddots&\vdots\\x_N^{(1)}&x_N^{(2)}&\cdots&x_N^{(n)}\end{bmatrix}-\begin{bmatrix}x_1^{(1)}&x_1^{(2)}&\cdots&x_1^{(n)}\\x_2^{(1)}&x_2^{(2)}&\cdots&x_2^{(n)}\\\vdots&\vdots&\ddots&\vdots\\x_N^{(1)}&x_N^{(2)}&\cdots&x_N^{(n)}\end{bmatrix}(\boldsymbol{w}_1,\boldsymbol{w}_2,\cdots,\boldsymbol{w}_d)\begin{bmatrix}\boldsymbol{w}_1^T\\\boldsymbol{w}_2^T\\\vdots\\\boldsymbol{w}_d^T\end{bmatrix}\\
&=\boldsymbol{X}^T-\begin{bmatrix}\boldsymbol{x}_1^T\boldsymbol{w}_1&\boldsymbol{x}_1^T\boldsymbol{w}_2&\cdots&\boldsymbol{x}_1^T\boldsymbol{w}_d\\\boldsymbol{x}_2^T\boldsymbol{w}_1&\boldsymbol{x}_2^T\boldsymbol{w}_2&\cdots&\boldsymbol{x}_2^T\boldsymbol{w}_d\\\vdots&\vdots&\ddots&\vdots\\\boldsymbol{x}_N^T\boldsymbol{w}_1&\boldsymbol{x}_N^T\boldsymbol{w}_2&\cdots&\boldsymbol{x}_N^T\boldsymbol{w}_d\end{bmatrix}\begin{bmatrix}\boldsymbol{w}_1^T\\\boldsymbol{w}_2^T\\\vdots\\\boldsymbol{w}_d^T\end{bmatrix}\end{aligned}$$

令 $\boldsymbol{w}_s = (w_s^{(1)}, w_s^{(2)}, \cdots, w_s^{(n)})^T$，则有：

$$\boldsymbol{X}^T \boldsymbol{W}_d \boldsymbol{W}_d^T = \begin{bmatrix} \sum_{s=1}^d \boldsymbol{x}_1^T \boldsymbol{w}_s w_s^{(1)} & \cdots & \sum_{s=1}^d \boldsymbol{x}_1^T \boldsymbol{w}_s w_s^{(n)} \\ \vdots & \ddots & \vdots \\ \sum_{s=1}^d \boldsymbol{x}_N^T \boldsymbol{w}_s w_s^{(1)} & \cdots & \sum_{s=1}^d \boldsymbol{x}_N^T \boldsymbol{w}_s w_s^{(n)} \end{bmatrix}$$

于是

$$\begin{aligned} ||\boldsymbol{X}^T - \boldsymbol{X}^T \boldsymbol{W}_d \boldsymbol{W}_d^T||_F^2 &= \sum_{i=1}^N \sum_{j=1}^n \left[x_i^{(j)} - \left(\sum_{s=1}^d \boldsymbol{x}_i^T \boldsymbol{w}_s w_s^{(j)} \right) \right]^2 \\ &= \sum_{i=1}^N ||\boldsymbol{x}_i - \sum_{s=1}^d (\boldsymbol{x}_i^T \boldsymbol{w}_s) \boldsymbol{w}_s||_2^2 \end{aligned}$$

要求解最优化问题：

$$\begin{aligned} \boldsymbol{W}_d^* &= \arg\min_{\boldsymbol{W}_d} ||\hat{\vec{\boldsymbol{x}}}_i - \vec{\boldsymbol{x}}_i||_2^2 \\ &= \arg\min_{\boldsymbol{W}_d} ||\boldsymbol{X}^T - \boldsymbol{X}^T \boldsymbol{W}_d \boldsymbol{W}_d^T||_F^2 \\ &= \arg\min_{\boldsymbol{W}_d} tr\left[(\boldsymbol{X}^T - \boldsymbol{X}^T \boldsymbol{W}_d \boldsymbol{W}_d^T)^T (\boldsymbol{X}^T - \boldsymbol{X}^T \boldsymbol{W}_d \boldsymbol{W}_d^T) \right] \\ &= \arg\min_{\boldsymbol{W}_d} tr\left[(\boldsymbol{X} - \boldsymbol{W}_d \boldsymbol{W}_d^T \boldsymbol{X})(\boldsymbol{X}^T - \boldsymbol{X}^T \boldsymbol{W}_d \boldsymbol{W}_d^T) \right] \\ &= \arg\min_{\boldsymbol{W}_d} tr\left[\boldsymbol{X}\boldsymbol{X}^T - \boldsymbol{X}\boldsymbol{X}^T \boldsymbol{W}_d \boldsymbol{W}_d^T - \boldsymbol{W}_d \boldsymbol{W}_d^T \boldsymbol{X}\boldsymbol{X}^T + \boldsymbol{W}_d \boldsymbol{W}_d^T \boldsymbol{X}\boldsymbol{X}^T \boldsymbol{W}_d \boldsymbol{W}_d^T \right] \\ &= \arg\min_{\boldsymbol{W}_d} \left[tr(\boldsymbol{X}\boldsymbol{X}^T) - tr(\boldsymbol{X}\boldsymbol{X}^T \boldsymbol{W}_d \boldsymbol{W}_d^T) - tr(\boldsymbol{W}_d \boldsymbol{W}_d^T \boldsymbol{X}\boldsymbol{X}^T) + tr(\boldsymbol{W}_d \boldsymbol{W}_d^T \boldsymbol{X}\boldsymbol{X}^T \boldsymbol{W}_d \boldsymbol{W}_d^T) \right] \end{aligned}$$

因为矩阵及其转置的迹相等，因此 $tr(\boldsymbol{X}\boldsymbol{X}^T \boldsymbol{W}_d \boldsymbol{W}_d^T) = tr(\boldsymbol{W}_d \boldsymbol{W}_d^T \boldsymbol{X}\boldsymbol{X}^T)$。由于可以在 $tr(\cdot)$ 中调整矩阵的顺序，则 $tr(\boldsymbol{W}_d \boldsymbol{W}_d^T \boldsymbol{X}\boldsymbol{X}^T \boldsymbol{W}_d \boldsymbol{W}_d^T) = tr(\boldsymbol{X}\boldsymbol{X}^T \boldsymbol{W}_d \boldsymbol{W}_d^T \boldsymbol{W}_d \boldsymbol{W}_d^T)$。

考虑到：

$$\boldsymbol{W}_d^T \boldsymbol{W}_d = \begin{bmatrix} \boldsymbol{w}_1^T \\ \boldsymbol{w}_2^T \\ \vdots \\ \boldsymbol{w}_d^T \end{bmatrix} (\boldsymbol{w}_1, \cdots, \boldsymbol{w}_d) = \boldsymbol{I}_{d \times d}$$

代入上式有：

$$tr(\boldsymbol{W}_d \boldsymbol{W}_d^T \boldsymbol{X}\boldsymbol{X}^T \boldsymbol{W}_d \boldsymbol{W}_d^T) = tr(\boldsymbol{X}\boldsymbol{X}^T \boldsymbol{W}_d \boldsymbol{W}_d^T)$$

于是：

$$\begin{aligned} \boldsymbol{W}_d^* &= \arg\min_{\boldsymbol{W}_d} \left[tr(\boldsymbol{X}\boldsymbol{X}^T) - 2tr(\boldsymbol{X}\boldsymbol{X}^T \boldsymbol{W}_d \boldsymbol{W}_d^T) + tr(\boldsymbol{X}\boldsymbol{X}^T \boldsymbol{W}_d \boldsymbol{W}_d^T) \right] \\ &= \arg\min_{\boldsymbol{W}_d} \left[tr(\boldsymbol{X}\boldsymbol{X}^T) - tr(\boldsymbol{X}\boldsymbol{X}^T \boldsymbol{W}_d \boldsymbol{W}_d^T) \right] \end{aligned}$$

由于 $tr(\boldsymbol{X}\boldsymbol{X}^T)$ 与 $\boldsymbol{W}_d$ 无关，因此：

$$\boldsymbol{W}_d^* = \arg\min_{\boldsymbol{W}_d} -tr(\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{W}_d\boldsymbol{W}_d^T) = \arg\max_{\boldsymbol{W}_d} tr(\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{W}_d\boldsymbol{W}_d^T)$$

调整矩阵顺序，则有：

$$\boldsymbol{W}_d^* = \arg\max_{\boldsymbol{W}_d} tr(\boldsymbol{W}_d^T\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{W}_d)$$

该最优化问题的求解就是求解 $\boldsymbol{X}\boldsymbol{X}^T$ 的特征值。因此只需要对矩阵 $\boldsymbol{X}\boldsymbol{X}^T$（也称为样本的协方差矩阵，它是一个 n 阶方阵）进行特征值分解，将求得的特征值排序：$\lambda_1 \geqslant \lambda_2 \geqslant \cdots \geqslant \lambda_n$，然后取前 d 个特征值对应的特征向量构成 $\boldsymbol{W} = (\boldsymbol{w}_1, \boldsymbol{w}_2, \cdots, \boldsymbol{w}_d)$。

当样本数据进行了中心化：即 $\sum_i \boldsymbol{x}_i = (0, 0, \cdots, 0)^T$ 时，$\sum_{i=1}^N \boldsymbol{x}_i\boldsymbol{x}_i^T = \boldsymbol{X}\boldsymbol{X}^T$ 就是样本集的协方差矩阵。

PCA 算法

PCA 算法

- 输入：样本集 $D = \{\vec{\boldsymbol{x}}_1, \vec{\boldsymbol{x}}_2, \cdots, \vec{\boldsymbol{x}}_N\}$；低维空间维数 d。
- 输出：投影矩阵 $\boldsymbol{W} = (\vec{\boldsymbol{w}}_1, \vec{\boldsymbol{w}}_2, \cdots, \vec{\boldsymbol{w}}_d)$。
- 算法步骤
  - 对所有样本进行中心化操作：

$$\vec{\boldsymbol{x}}_i \leftarrow \vec{\boldsymbol{x}}_i - \frac{1}{N}\sum_{j=1}^N \vec{\boldsymbol{x}}_j$$

  - 计算样本的协方差矩阵 $\boldsymbol{X}\boldsymbol{X}^T$；
  - 对协方差矩阵 $\boldsymbol{X}\boldsymbol{X}^T$ 做特征值分解；
  - 取最大的 d 个特征值对应的特征向量 $\vec{\boldsymbol{w}}_1, \vec{\boldsymbol{w}}_2, \cdots, \vec{\boldsymbol{w}}_d$，构造投影矩阵 $\boldsymbol{W} = (\vec{\boldsymbol{w}}_1, \vec{\boldsymbol{w}}_2, \cdots, \vec{\boldsymbol{w}}_d)$。

通常低维空间维数 d 的选取有两种方法：

- 通过交叉验证法选取较好的 d（在降维后的学习器的性能比较好）。
- 从算法原理的角度设置一个阈值，比如 $t = 95\%$，然后选取使得下式成立的最小的 d 的值：

$$\frac{\sum_{i=1}^{d} \lambda_i}{\sum_{i=1}^{n} \lambda_i} \geqslant t$$

  其中 λ_i 从大到小排列。

PCA 降维的两个准则

PCA降维的准则有以下两个。

- 最近重构性：就是前面介绍的样本集中所有点，重构后的点距离原来的点的误差之和最小。
- 最大可分性：样本点在低维空间的投影尽可能分开。

可以证明，最近重构性就等价于最大可分性。证明如下：对于样本点 $\vec{\boldsymbol{x}}_i$，它在降维后空间中的投影是 $\vec{\boldsymbol{z}}_i$。根据：

$$\hat{\vec{\boldsymbol{x}}}_i = (\vec{\boldsymbol{w}}_1, \vec{\boldsymbol{w}}_2, \cdots, \vec{\boldsymbol{w}}_d) \begin{bmatrix} z_i^{(1)} \\ z_i^{(2)} \\ \vdots \\ z_i^{(d)} \end{bmatrix} = \boldsymbol{W} \vec{\boldsymbol{z}}_i$$

由投影矩阵的性质，以及 $\hat{\vec{\boldsymbol{x}}}_i$ 与 $\vec{\boldsymbol{x}}_i$ 的关系，则有：$\vec{\boldsymbol{z}}_i = \boldsymbol{W}^T \vec{\boldsymbol{x}}_i$。

由于样本数据进行了中心化：即 $\sum_i \vec{\boldsymbol{x}}_i = (0, 0, \cdots, 0)^T$，故投影后，样本点的方差为

$$\sum_{i=1}^{N} \boldsymbol{W}^T \vec{\boldsymbol{x}}_i \vec{\boldsymbol{x}}_i^T \boldsymbol{W}$$

令 $\boldsymbol{X} = (\vec{\boldsymbol{x}}_1, \vec{\boldsymbol{x}}_2, \cdots, \vec{\boldsymbol{x}}_N)$ 为 $n \times N$ 维矩阵，于是根据样本点的方差最大，优化目标可写为：

$$\max_{\boldsymbol{W}} tr(\boldsymbol{W}^T \boldsymbol{X} \boldsymbol{X}^T \boldsymbol{W}) s.t. \boldsymbol{W}^T \boldsymbol{W} = \boldsymbol{I}$$

这就是前面最近重构性推导的结果。

5.2.3 SVD 降维

奇异值分解（SVD）：设 $\boldsymbol{X}$ 为 $n \times N$ 阶矩阵，且 $rank(\boldsymbol{X}) = r$，则存在 n 阶正交矩阵 $\boldsymbol{V}$ 和 N 阶正交矩阵 $\boldsymbol{V}$，使得：

$$\boldsymbol{V}^T \boldsymbol{X} \boldsymbol{U} = \begin{bmatrix} \Sigma & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}_{n \times N}$$

其中

$$\Sigma = \begin{bmatrix} \sigma_1 & 0 & 0 & \cdots & 0 \\ 0 & \sigma_2 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \sigma_r \end{bmatrix}$$

其中，$\sigma_1 \geqslant \sigma_2 \geqslant \cdots \geqslant \sigma_r > 0$

根据正交矩阵的性质，$\boldsymbol{V}\boldsymbol{V}^T = \boldsymbol{I}, \boldsymbol{U}\boldsymbol{U}^T = \boldsymbol{I}$，有：

$$\boldsymbol{X} = \boldsymbol{V} \begin{bmatrix} \Sigma & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}_{n \times N} \boldsymbol{U}^T \Longrightarrow \boldsymbol{X}^T = \boldsymbol{U} \begin{bmatrix} \Sigma & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}_{N \times n} \boldsymbol{V}^T$$

则有 $\boldsymbol{X}\boldsymbol{X}^T = \boldsymbol{V}\boldsymbol{M}\boldsymbol{V}^T$，其中 $\boldsymbol{M}$ 是个 n 阶对角矩阵：

$$\boldsymbol{M} = \begin{bmatrix} \Sigma & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}_{n \times N} \begin{bmatrix} \Sigma & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}_{N \times n} = \begin{bmatrix} \lambda_1 & 0 & 0 & \cdots & 0 \\ 0 & \lambda_2 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \lambda_n \end{bmatrix}_{n \times n}$$

$$\lambda_i = \sigma_i^2 \quad i = 1, 2, \cdots, r$$

$$\lambda_i = 0 \quad i = r+1, r+2, \cdots, n$$

于是有：$\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{V} = \boldsymbol{V}\boldsymbol{M}$。根据 $\boldsymbol{M}$ 是个对角矩阵的性质，有：$\boldsymbol{V}\boldsymbol{M} = \boldsymbol{M}\boldsymbol{V}$，则有：

$$\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{V} = \boldsymbol{M}\boldsymbol{V}$$

则 $\lambda_i, i = 1, 2, \cdots, r$ 就是 $\boldsymbol{X}\boldsymbol{X}^T$ 的特征值，其对应的特征向量组成正交矩阵 $\boldsymbol{V}$。因此SVD奇异值分解等价于PCA主成分分析，核心都是求解 $\boldsymbol{X}\boldsymbol{X}^T$ 的特征值以及对应的特征向量。

5.2.4 核化线性（KPCA）降维

PCA方法假设从高维空间到低维空间的函数映射是线性的，但是在不少现实任务中，可能需要非线性映射才能找到合适的低维空间来降维。非线性降维的一种常用方法是基于核技

巧对线性降维方法进行核化（kernelized）。核主成分分析（Kernelized PCA，KPCA）是对PCA的一种推广。

假定原始属性空间中的样本点 $\vec{\boldsymbol{x}}_i$ 通过将 ϕ 映射到高维特征空间的坐标为 $\vec{\boldsymbol{x}}_{i,\phi}$，即 $\vec{\boldsymbol{x}}_{i,\phi} = \phi(\vec{\boldsymbol{x}}_i)$。且假设高维特征空间是 n 维的，即：$\vec{\boldsymbol{x}}_{i,\phi} \in \mathbb{R}^n$。

假定要将高维特征空间中的数据投影到低维空间中，投影矩阵 $\boldsymbol{W}$ 为 $n \times d$ 维矩阵，根据 PCA 推导的结果，要求解方程：

$$\boldsymbol{X}_\phi \boldsymbol{X}_\phi^T \boldsymbol{W} = \lambda \boldsymbol{W}$$

其中 $\boldsymbol{X}_\phi = (\vec{\boldsymbol{x}}_{1,\phi}, \vec{\boldsymbol{x}}_{2,\phi}, \cdots, \vec{\boldsymbol{x}}_{N,\phi})$ 为 $n \times N$ 维矩阵，于是有：

$$\left(\sum_{i=1}^{N} \phi(\vec{\boldsymbol{x}}_i)\phi(\vec{\boldsymbol{x}}_i)^T\right)\boldsymbol{W} = \lambda \boldsymbol{W}$$

通常并不清楚 ϕ 的解析表达式，于是引入核函数：

$$\kappa(\vec{\boldsymbol{x}}_i, \vec{\boldsymbol{x}}_j) = \phi(\vec{\boldsymbol{x}}_i)^T \phi(\vec{\boldsymbol{x}}_j)$$

- 定义核矩阵：

$$\boldsymbol{K} = \begin{bmatrix} \kappa(\vec{\boldsymbol{x}}_1, \vec{\boldsymbol{x}}_1) & \kappa(\vec{\boldsymbol{x}}_1, \vec{\boldsymbol{x}}_2) & \cdots, \kappa(\vec{\boldsymbol{x}}_1, \vec{\boldsymbol{x}}_N) \\ \kappa(\vec{\boldsymbol{x}}_2, \vec{\boldsymbol{x}}_1) & \kappa(\vec{\boldsymbol{x}}_2, \vec{\boldsymbol{x}}_2) & \cdots, \kappa(\vec{\boldsymbol{x}}_2, \vec{\boldsymbol{x}}_N) \\ \vdots & \vdots & \ddots, \vdots \\ \kappa(\vec{\boldsymbol{x}}_N, \vec{\boldsymbol{x}}_1) & \kappa(\vec{\boldsymbol{x}}_N, \vec{\boldsymbol{x}}_2) & \cdots, \kappa(\vec{\boldsymbol{x}}_N, \vec{\boldsymbol{x}}_N) \end{bmatrix}$$

  则有：$\boldsymbol{X}_\phi^T \boldsymbol{X}_\phi = \boldsymbol{K}$。
- 定义：

$$\vec{\alpha}_i = \frac{\vec{\boldsymbol{x}}_{i,\phi}^T \boldsymbol{W}}{\lambda}$$

  则 $\vec{\alpha}_i$ 为 $1 \times d$ 维行向量。
- 定义：$\boldsymbol{A} = (\vec{\alpha}_1, \vec{\alpha}_2, \cdots, \vec{\alpha}_N)^T$ 为 $N \times d$ 维矩阵

  则有：

$$\boldsymbol{W} = \frac{1}{\lambda}\left(\sum_{i=1}^{N} \vec{\boldsymbol{x}}_{i,\phi}\vec{\boldsymbol{x}}_{i,\phi}^T\right)\boldsymbol{W} = \sum_{i=1}^{N} \vec{\boldsymbol{x}}_{i,\phi}\frac{\vec{\boldsymbol{x}}_{i,\phi}^T \boldsymbol{W}}{\lambda} = \sum_{i=1}^{N} \vec{\boldsymbol{x}}_{i,\phi}\vec{\alpha}_i = \boldsymbol{X}_\phi \boldsymbol{A}$$

将 $\boldsymbol{W} = \boldsymbol{X}_\phi \boldsymbol{A}$ 代入

$$\boldsymbol{X}_\phi \boldsymbol{X}_\phi^T \boldsymbol{W} = \lambda \boldsymbol{W}$$

得到：

$$\boldsymbol{X}_\phi \boldsymbol{X}_\phi^T \boldsymbol{X}_\phi \boldsymbol{A} = \lambda \boldsymbol{X}_\phi \boldsymbol{A}$$

两边同时左乘以 $\boldsymbol{X}_\phi^T$，再代入 $\boldsymbol{X}_\phi^T\boldsymbol{X}_\phi=\boldsymbol{K}$ 有：

$$\boldsymbol{KKA}=\lambda\boldsymbol{KA}$$

若要求核矩阵可逆，则上式两边同时左乘以 $\boldsymbol{K}^{-1}$，则有：

$$\boldsymbol{KA}=\lambda\boldsymbol{A}$$

同样该问题也是一个特征值分解问题。取 $\boldsymbol{K}$ 最大的 d 个特征值对应的特征向量组成 $\boldsymbol{W}$ 即可。

对于新样本 $\vec{\boldsymbol{x}}$，其投影后第 $j,j=1,2,\cdots,d$ 维的坐标为：

$$z_j=\boldsymbol{w}_j^T\phi(\vec{\boldsymbol{x}})=\sum_{i=1}^N\alpha_i^{(j)}\phi(\vec{\boldsymbol{x}}_i)^T\phi(\vec{\boldsymbol{x}})=\sum_{i=1}^N\alpha_i^{(j)}\kappa(\vec{\boldsymbol{x}}_i,\vec{\boldsymbol{x}})$$

其中，$\alpha_i^{(j)}$ 为行向量 $\vec{\alpha}_i$ 的第 j 个分量。可以看到，为了获取投影后的坐标，KPCA需要对所有样本求和，因此它的计算开销较大。

5.2.5 流形学习降维

流形学习（manifold learning）是一类借鉴了拓扑流形概念的降维方法。流形是局部和欧氏空间同胚的空间：它在局部具有欧氏空间的性质，能用欧氏距离进行距离计算。流形学习要想取得很好的效果，要求样本数据比较密集，因此流形学习方法在实践中的降维性能往往没有预期的好。

5.2.6 多维缩放（MDS）降维

MDS 原理

多维缩放（Multiple Dimensional Scaling，MDS）要求原始空间中样本之间的距离在低维空间中得到保持。

假设 N 个样本在原始空间中的距离矩阵为 $\boldsymbol{D}=(d_{ij})_{N\times N}$：

$$\boldsymbol{D}=\begin{bmatrix}d_{1,1} & d_{1,2} & \cdots & d_{1,N}\\ d_{2,1} & d_{2,2} & \cdots & d_{2,N}\\ \vdots & \vdots & \ddots & \vdots\\ d_{N,1} & d_{N,2} & \cdots & d_{N,N}\end{bmatrix}$$

其中，$d_{ij} = ||\vec{\boldsymbol{x}}_i - \vec{\boldsymbol{x}}_j||$ 为样本 $\vec{\boldsymbol{x}}_i$ 到样本 $\vec{\boldsymbol{x}}_j$ 的距离。

假设原始样本是在 n 维空间，我们的目标是在 $n', n' < n$ 维空间里获取样本，欧氏距离保持不变。

假设样本集在原空间的表示 $\boldsymbol{X} = (\vec{\boldsymbol{x}}_1, \vec{\boldsymbol{x}}_2, \cdots, \vec{\boldsymbol{x}}_N)$ 为 $n \times N$ 维矩阵，样本集在降维后空间的坐标 $\boldsymbol{Z} = (\vec{\boldsymbol{z}}_1, \vec{\boldsymbol{z}}_2, \cdots, \vec{\boldsymbol{z}}_N)$ 为 $n' \times N$ 维矩阵。我们所求的正是 $\boldsymbol{Z}$ 矩阵。

> 同时我们也不知道 n'。很明显：并不是所有的低维空间在满足线性映射之后，样本点之间的欧氏距离都保持不变。

令 $\boldsymbol{B} = \boldsymbol{Z}^T\boldsymbol{Z}$ 为 $N \times N$ 维矩阵，即

$$\boldsymbol{B} = \begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,N} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ b_{N,1} & b_{N,2} & \cdots & b_{N,N} \end{bmatrix}$$

其中 $b_{i,j} = \vec{\boldsymbol{z}}_i \cdot \vec{\boldsymbol{z}}_j$ 为降维后样本的内积。

则根据降维前后样本的欧氏距离保持不变，有：

$$d_{ij}^2 = ||\vec{\boldsymbol{z}}_i - \vec{\boldsymbol{z}}_j||^2 = ||\vec{\boldsymbol{z}}_i||^2 + ||\vec{\boldsymbol{z}}_j||^2 - 2\vec{\boldsymbol{z}}_i^T\vec{\boldsymbol{z}}_j = b_{i,i} + b_{j,j} - 2b_{i,j}$$

假设降维后的样本集 $\boldsymbol{Z}$ 被中心化，即 $\sum_{i=1}^{N} \vec{\boldsymbol{z}}_i = \vec{\boldsymbol{0}}$，则矩阵 $\boldsymbol{B}$ 的每行之和均为零，每列之和均为零，即：

$$\sum_{i=1}^{N} b_{i,j} = 0, \quad j = 1, 2, \cdots, N$$
$$\sum_{j=1}^{N} b_{i,j} = 0, \quad i = 1, 2, \cdots, N$$

于是有：

$$\sum_{i=1}^{N} d_{ij}^2 = \sum_{i=1}^{N} b_{i,i} + Nb_{j,j} = tr(\boldsymbol{B}) + Nb_{j,j}$$
$$\sum_{j=1}^{N} d_{ij}^2 = \sum_{j=1}^{N} b_{j,j} + Nb_{i,i} = tr(\boldsymbol{B}) + Nb_{i,i}$$
$$\sum_{i=1}^{N}\sum_{j=1}^{N} d_{ij}^2 = \sum_{i=1}^{N} \left(tr(\boldsymbol{B}) + Nb_{i,i}\right) = 2Ntr(\boldsymbol{B})$$

其中，$tr(\boldsymbol{B})$ 表示矩阵 $\boldsymbol{B}$ 的迹。

令：

$$d_{i,\cdot}^2 = \frac{1}{N}\sum_{j=1}^{N} d_{ij}^2 = \frac{tr(\boldsymbol{B})}{N} + b_{i,i}$$

$$d_{j,\cdot}^2 = \frac{1}{N}\sum_{i=1}^{N} d_{ij}^2 = \frac{tr(\boldsymbol{B})}{N} + b_{j,j}$$

$$d_{\cdot,\cdot}^2 = \frac{1}{N^2}\sum_{i=1}^{N}\sum_{j=1}^{N} d_{ij}^2 = \frac{2tr(\boldsymbol{B})}{N}$$

代入 $d_{ij}^2 = b_{i,i} + b_{j,j} - 2b_{i,j}$，有：

$$b_{i,j} = \frac{b_{i,i} + b_{j,j} - d_{ij}^2}{2} = \frac{d_{i,\cdot}^2 + d_{j,\cdot}^2 - d_{\cdot,\cdot}^2 - d_{ij}^2}{2}$$

右式根据 d_{ij} 给出了 $b_{i,j}$，因此可以根据原始空间中的距离矩阵 $\boldsymbol{D}$ 求出在降维后空间的内积矩阵 $\boldsymbol{B}$。现在的问题是，已知内积矩阵 $\boldsymbol{B} = \boldsymbol{Z}^T\boldsymbol{Z}$，如何求得矩阵 $\boldsymbol{Z}$。

对矩阵 $\boldsymbol{B}$ 做特征值分解，设 $\boldsymbol{B} = \boldsymbol{V}\Lambda\boldsymbol{V}^T$，其中 $\Lambda = diag(\lambda_1, \lambda_2, \cdots, \lambda_N)$ 为特征值构成的对角矩阵，其中 $\lambda_1 \geqslant \lambda_2 \geqslant \cdots \geqslant \lambda_N$，$\boldsymbol{V}$ 为特征向量矩阵。

假定特征值中有 n^* 个非零特征值，它们构成对角矩阵 $\Lambda_* = diag(\lambda_1, \lambda_2, \cdots, \lambda_{n^*})$。令 $\boldsymbol{V}_*$ 为对应的特征向量矩阵，则

$$\boldsymbol{Z} = \Lambda_*^{1/2}\boldsymbol{V}_*^T$$

其中，$\boldsymbol{Z}$ 为 $n^* \times N$ 阶矩阵。此时有 $n' = n^*$。

在现实应用中，为了有效降维，往往仅需要降维后的距离与原始空间中的距离尽可能相等，而不必严格相等。此时可以取 $n' \ll n^* \leqslant n$ 个最大特征值构成的对角矩阵为：

$$\tilde{\Lambda} = diag(\lambda_1, \lambda_2, \cdots, \lambda_{n'})$$

令 $\tilde{\boldsymbol{V}}$ 表示对应的特征向量矩阵，则

$$\boldsymbol{Z} = \tilde{\Lambda}^{1/2}\tilde{\boldsymbol{V}}^T \in \mathbb{R}^{n' \times N}$$

MDS 算法

多维缩放（MDS）算法如下。

- 输入：距离矩阵 $\boldsymbol{D} \in \mathbb{R}^{N \times N}$；低维空间维数 n'。
- 输出：样本集在低维空间中的矩阵 $\boldsymbol{Z}$。

- 算法步骤
  - 根据下列式子计算 $d_{i,\cdot}^2, d_{j,\cdot}^2, d_{\cdot,\cdot}^2$

$$d_{i,\cdot}^2 = \frac{1}{N}\sum_{j=1}^{N} d_{ij}^2 = \frac{tr(\boldsymbol{B})}{N} + b_{i,i}$$

$$d_{j,\cdot}^2 = \frac{1}{N}\sum_{i=1}^{N} d_{ij}^2 = \frac{tr(\boldsymbol{B})}{N} + b_{j,j}$$

$$d_{\cdot,\cdot}^2 = \frac{1}{N^2}\sum_{i=1}^{N}\sum_{j=1}^{N} d_{ij}^2 = \frac{2tr(\boldsymbol{B})}{N}$$

  - 根据下式计算矩阵 $\boldsymbol{B}$：

$$b_{i,j} = \frac{b_{i,i} + b_{j,j} - d_{ij}^2}{2} = \frac{d_{i,\cdot}^2 + d_{j,\cdot}^2 - d_{\cdot,\cdot}^2 - d_{ij}^2}{2}$$

  - 对矩阵 $\boldsymbol{B}$ 进行特征值分解。
  - 取 $\tilde{\Lambda}$ 为 n' 个最大特征值所构成的对角矩阵，$\tilde{\boldsymbol{V}}$ 表示对应的特征向量矩阵，则：

$$\boldsymbol{Z} = \tilde{\Lambda}^{1/2}\tilde{\boldsymbol{V}}^T \in \mathbb{R}^{n' \times N}$$

5.2.7 等度量映射（Isomap）降维

等度量映射（Isometric Mapping，Isomap）原理如下。

- 首先建立近邻连接图：利用流形在局部上与欧氏空间同胚这个性质，基于欧氏距离，对每个点找出它在低维流形上的近邻点建立近邻连接图。
- 计算任意两点之间的距离：计算近邻连接图上任意两点之间的最短路径问题，作为两点之间的距离。
- 在得到任意两点的距离之后，就可以通过MDS算法来获得样本点在低维空间中的坐标。

Isomap算法如下。

- 输入：样本集 $D = \{\vec{\boldsymbol{x}}_1, \vec{\boldsymbol{x}}_2, \cdots, \vec{\boldsymbol{x}}_N\}$；近邻参数 k；低维空间维数 n'。
- 输出：样本集在低维空间中的矩阵 $\boldsymbol{Z}$。
- 算法步骤
  - 对每个样本点 $\vec{\boldsymbol{x}}_i$，计算它的 k 近邻；同时将 $\vec{\boldsymbol{x}}_i$ 与它的 k 近邻的距离设置为欧氏距离，与其他点的距离设置为无穷大。
  - 调用最短路径算法计算任意两个样本点之间的距离，获得距离矩阵 $\boldsymbol{D} \in \mathbb{R}^{N \times N}$。
  - 调用多维缩放MDS算法，获得样本集在低维空间中的矩阵 $\boldsymbol{Z}$。

Isomap算法有个很大的问题：对于新样本，难以将其映射到低维空间。理论上可以将新样本添加到样本集中，重新调用Isomap算法，这种方案计算量太大。一般的解决方法是：训练一个回归学习器来对新样本的低维空间进行预测。

近邻图有如下两种类型。

- k 近邻图：指定近邻点个数，如指定距离最近的 k 个点为近邻点。
- ϵ 近邻图：指定距离阈值 ϵ ，距离小于 ϵ 的点被认为是近邻点。

在建立近邻图的时候要注意控制近邻图的范围，否则容易出现“短路”或者“断路”问题：

- “短路”问题：近邻范围指定过大，距离很远的点也被误认为近邻。
- “断路”问题：近邻范围指定过小，图中本应该相连的区域被认为是断开的。

5.2.8 局部线性嵌入（LLE）

LLE 原理

局部线性嵌入（Locally Linear Embedding，LLE）的目标是：保持邻域内样本之间的线性关系。

对每个样本 $\vec{\mathbf{x}}_i$，首先寻找其近邻点，假设这些近邻点的下标集合为 Q_i。然后需要计算基于 $\vec{\mathbf{x}}_i$ 的近邻点对 $\vec{\mathbf{x}}_i$ 进行线性重构的系数 $\vec{\mathbf{w}}_i$ 。定义样本集重构误差为：

$$err = \sum_{i=1}^{N} ||\vec{\mathbf{x}}_i - \sum_{j\in Q_i} w_{i,j}\vec{\mathbf{x}}_j||_2^2$$

其中，$w_{i,j}$ 为 $\vec{\mathbf{w}}_i$ 的分量。我们的目标是样本集重构误差最小，即：

$$\min_{\vec{\mathbf{w}}_1,\vec{\mathbf{w}}_2,\cdots,\vec{\mathbf{w}}_N} \sum_{i=1}^{N} ||\vec{\mathbf{x}}_i - \sum_{j\in Q_i} w_{i,j}\vec{\mathbf{x}}_j||_2^2$$

这样的解有无数个，对权重进行归一化处理，即：

$$\sum_{j\in Q_i} w_{i,j} = 1, i = 1,2,\cdots,N$$

这样一来，就是求解最优化问题：

$$\min_{\vec{\boldsymbol{w}}_1,\vec{\boldsymbol{w}}_2,\cdots,\vec{\boldsymbol{w}}_N} \sum_{i=1}^{N} ||\vec{\boldsymbol{x}}_i - \sum_{j\in Q_i} w_{i,j}\vec{\boldsymbol{x}}_j||_2^2$$
$$s.t. \quad \sum_{j\in Q_i} w_{i,j} = 1, i = 1, 2, \cdots, N$$

该最优化问题有解析解。令 $C_{j,k} = (\vec{\boldsymbol{x}}_i - \vec{\boldsymbol{x}}_j)^T(\vec{\boldsymbol{x}}_i - \vec{\boldsymbol{x}}_k)$，则可以解出：

$$w_{i,j} = \frac{\sum_{k\in Q_i} C_{j,k}^{-1}}{\sum_{l,s\in Q_i} C_{l,s}^{-1}}, j \in Q_i$$

求出了线性重构的系数 $\vec{\boldsymbol{w}}_i$ 之后，LLE在低维空间中保持 $\vec{\boldsymbol{w}}_i$ 不变。设 $\vec{\boldsymbol{x}}_i$ 对应的低维坐标 $\vec{\boldsymbol{z}}_i$，已知线性重构的系数 $\vec{\boldsymbol{w}}_i$，定义样本集在低维空间中重构误差为：

$$err' = \sum_{i=1}^{N} ||\vec{\boldsymbol{z}}_i - \sum_{j\in Q_i} w_{i,j}\vec{\boldsymbol{z}}_j||_2^2$$

现在的问题是要求出 $\vec{\boldsymbol{z}}_i$，从而使上式最小。即求解：

$$\min_{\vec{\boldsymbol{z}}_1,\vec{\boldsymbol{z}}_2,\cdots,\vec{\boldsymbol{z}}_N} \sum_{i=1}^{N} ||\vec{\boldsymbol{z}}_i - \sum_{j\in Q_i} w_{i,j}\vec{\boldsymbol{z}}_j||_2^2$$

令 $\boldsymbol{Z} = (\vec{\boldsymbol{z}}_1, \vec{\boldsymbol{z}}_2, \cdots, \vec{\boldsymbol{z}}_N) \in \mathbb{R}^{n'\times N}$，其中 n' 为低维空间的维数（n 为原始样本所在的高维空间的维数）。令：

$$\boldsymbol{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,N} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ w_{N,1} & w_{N,2} & \cdots & w_{N,N} \end{bmatrix}$$

定义 $\boldsymbol{M} = (\boldsymbol{I} - \boldsymbol{W})^T(\boldsymbol{I} - \boldsymbol{W})$，于是最优化问题可重写为：

$$\min_{\boldsymbol{Z}} tr(\boldsymbol{Z}\boldsymbol{M}\boldsymbol{Z}^T)$$

该最优化问题有无数个解。添加约束 $\boldsymbol{Z}\boldsymbol{Z}^T = \boldsymbol{I}$，于是最优化问题为：

$$\min_{\boldsymbol{Z}} tr(\boldsymbol{Z}\boldsymbol{M}\boldsymbol{Z}^T)$$
$$s.t. \quad \boldsymbol{Z}\boldsymbol{Z}^T = \boldsymbol{I}$$

该最优化问题可以通过特征值分解求解：选取 $\boldsymbol{M}$ 最小的 n' 个特征值对应的特征向量组成的矩阵即为 $\boldsymbol{Z}^T$。

LLE 算法

LLE算法如下。

- 输入：样本集 $D=\{\vec{\boldsymbol{x}}_1,\vec{\boldsymbol{x}}_2,\cdots,\vec{\boldsymbol{x}}_N\}$；近邻参数 k；低维空间维数 n'。
- 输出：样本集在低维空间中的矩阵 $\boldsymbol{Z}$。
- 算法步骤
  - 对于样本集中的每个点 $\vec{\boldsymbol{x}}_i, i=1,2,\cdots,N$，执行下列操作：
    - 确定 $\vec{\boldsymbol{x}}_i$ 的 k 近邻，获得其近邻下标集合 Q_i。
    - 对于 $j\in Q_i$，根据下式计算 $w_{i,j}$

$$w_{i,j}=\frac{\sum_{k\in Q_i}C_{j,k}^{-1}}{\sum_{l,s\in Q_i}C_{l,s}^{-1}}$$
$$C_{j,k}=(\vec{\boldsymbol{x}}_i-\vec{\boldsymbol{x}}_j)^T(\vec{\boldsymbol{x}}_i-\vec{\boldsymbol{x}}_k)$$

    - 对于 $j\notin Q_i$，$w_{i,j}=0$。
  - 根据 $w_{i,j}$ 构建矩阵 $\boldsymbol{W}$。
  - 计算 $\boldsymbol{M}=(\boldsymbol{I}-\boldsymbol{W})^T(\boldsymbol{I}-\boldsymbol{W})$。
  - 对 $\boldsymbol{M}$ 进行特征值分解，取其最小的 n' 个特征值对应的特征向量，即得到样本集在低维空间中的矩阵 $\boldsymbol{Z}$。

5.3 Python 实战

首先导入包

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import   datasets,decomposition,manifold
```

使用scikit-learn自带的鸢尾花数据集，给出加载数据集的函数如下：

```
def load_data():
    iris=datasets.load_iris()
    return  iris.data,iris.target
```

注意：数据降维并没有一个好坏标准，因此这里仅给出降维的一些结果以及可视化降维之后的数据集。

PCA

scikit-learn中提供了一个PCA类来实现 PCA 模型，其原型为：

```
class sklearn.decomposition.PCA(n_components=None, copy=True, whiten=False)
```

参数

- n_components：一个整数，指定降维后的维数。
  - 如果为None，则选择它的值为 min(n_samples,n_features)。
  - 如果为字符串'mle'，则使用Minka's MLE算法来猜测降维后的维数。
  - 如果为大于 0、小于 1 的浮点数，则指定的是降维后的维数占原始维数的百分比。
- copy：一个布尔值。如果为False，则直接使用原始数据来训练，结果会覆盖原始数据所在的数组。
- whiten：一个布尔值。如果为True，则会将特征向量除以 n_samples倍的特征值，从而保证非相关输出的方差为 1。

> 白化操作可能会丢弃部分信息，但是它有时候在接下来的学习器学习阶段能获得更加好的性能。

属性

- components_：主成分数组。
- explained_variance_ratio_：一个数组，元素是每个主成分的explained variance的比例。
- mean_：一个数组，元素是每个特征的统计平均值。
- n_components_：一个整数，指示主成分有多少个元素。

方法

- fit(X[, y])：训练模型。
- transform(X)：执行降维。
- fit_transform(X[, y])：训练模型并且降维。
- inverse_transform(X)：执行升维（逆向操作），将数据从低维空间逆向转换到原始空间。

注意：decomposition.PCA基于scipy.linalg来实现SVD分解，因此它不能应用于稀疏矩阵，并且无法适用于超大规模数据（因它要求所有的数据一次加载进内存）。

首先使用PCA类，给出测试函数：

```
def test_PCA(*data):
    X,y=data
    pca=decomposition.PCA(n_components=None)
    pca.fit(X)
    print('explained variance ratio : %s'% str(pca.explained_variance_ratio_))
```

然后调用test_PCA函数：

```
X,y=load_data()
test_PCA(X,y)
```

运行结果如下：

```
explained variance ratio : [ 0.92461621  0.05301557  0.01718514  0.00518309]
```

可以看到，四个特征值的比例分别占比 0.92461621、0.05301557、0.01718514、0.00518309。因此可以将其原始特征（4 维）降低到 2 维。

给出绘制降维后的样本分布图的函数：

```
def plot_PCA(*data):
    X,y=data
    pca=decomposition.PCA(n_components=2)
    pca.fit(X)
    X_r=pca.transform(X)

    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    colors=((1,0,0),(0,1,0),(0,0,1),(0.5,0.5,0),(0,0.5,0.5),(0.5,0,0.5),
        (0.4,0.6,0),(0.6,0.4,0),(0,0.6,0.4),(0.5,0.3,0.2),)
    for label ,color in zip( np.unique(y),colors):
        position=y==label
        ax.scatter(X_r[position,0],X_r[position,1],label="target= %d"%label,color=color)

    ax.set_xlabel("X[0]")
    ax.set_ylabel("Y[0]")
    ax.legend(loc="best")
    ax.set_title("PCA")
    plt.show()
```

调用plot_PCA函数，图形如图 5.1 所示。

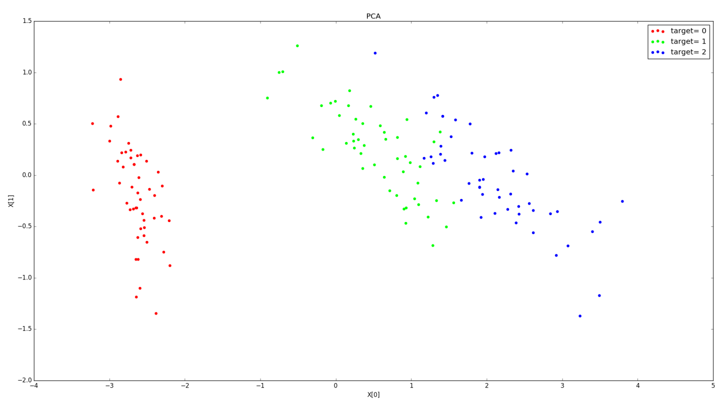


图 5.1 PCA

超大规模数据降维 IncrementalPCA

IncrementalPCA 可以适用于超大规模数据，它可以将数据分批加载进内存。其原型为：

```
class sklearn.decomposition.IncrementalPCA(n_components=None, whiten=False, copy=True,
 batch_size=None)
```

参数

- n_components：一个整数，指定降维后的维数。
  - 如果为None，则选择它的值为 min(n_samples,n_features)。
- batch_size：一个整数或者None，指定每个批次训练时，使用的样本数量。只有当调用fit()/partial_fit()方法时会用到该参数。
  - 如果为None，则由算法自动推断。
- copy：一个布尔值。如果为False，则直接使用原始数据来训练，结果会覆盖原始数据所在的数组。
- whiten：一个布尔值。如果为True，则会将特征向量除以 n_samples倍的特征值，从而保证非相关输出的方差为 1。

> 白化操作可能会丢弃部分信息，但是在接下来的学习器学习阶段它有时候能获得更加好的性能。

属性

- components_：主成分数组。
- explained_variance_：每个成分对应的 explained variance。
- explained_variance_ratio_：一个数组，元素是每个主成分的explained variance的比例。
- mean_：一个数组，元素是每个特征的统计平均值。每调用一次partial_fit()方法，就会更新一次该属性。
- var_：一个数组，元素是每个特征的经验方差。每调用一次partial_fit()方法，就会更新一次该属性。
- n_components_：一个整数，指示主成分有多少个元素。
- n_samples_seen_：一个整数，指示目前已经处理了多少个样本。每调用一次partial_fit()方法，就会更新一次该属性。每调用一次fit()方法，就会清零该属性。

方法

- fit(X[, y])：训练模型，使用batch_size个样本。
- partial_fit(X[, y, check_input])：继续训练模型，使用batch_size个样本。
- transform(X)：执行降维。
- fit_transform(X[, y])：训练模型并且降维。

- inverse_transform(X)：执行升维（逆向操作），将数据从低维空间逆向转换到原始空间。

由于IncrementalPCA的接口与用法几乎与PCA完全一致，因此示例可参考PCA的示例。

KernelPCA

KernelPCA是scikit-learn实现的核化PCA模型，其原型为：

```
class sklearn.decomposition.KernelPCA(n_components=None, kernel='linear', gamma=None,
degree=3, coef0=1, kernel_params=None, alpha=1.0, fit_inverse_transform=False,
 eigen_solver='auto', tol=0, max_iter=None, remove_zero_eig=False)
```

参数

- n_components：一个整数，指定降维后的维数。
  - 如果为None，则维数不变。
- kernel：一个字符串，指定核函数。
  - 'linear'：线性核，$K(\vec{x},\vec{z}) = \vec{x}\cdot\vec{z}$。
  - 'poly'：多项式核，$K(\vec{x},\vec{z}) = (\gamma(\vec{x}\cdot\vec{z}+1)+r)^p$，其中 p 由 degree参数决定，γ 由gamma参数决定，r 由 coef0参数决定。
  - 'rbf'（默认值）：高斯核函数，$K(\vec{x},\vec{z}) = \exp(-\gamma||\vec{x}-\vec{z}||^2)$，其中 γ 由 gamma参数决定。
  - 'sigmoid'：$K(\vec{x},\vec{z}) = \tanh(\gamma(\vec{x}\cdot\vec{z})+r)$。其中 γ 由 gamma参数决定，r由 coef0参数决定。
  - 'precomputed'：表示提供了kernel matrix。
  - 或者提供了一个可调用对象，该对象用于计算kernel matrix。
- degree：一个整数。当核函数是多项式核函数时，指定多项式的系数。对于其他核函数，该参数无效。
- gamma：一个浮点数。当核函数是'rbf'、'poly'、'sigmoid'时，为核函数的系数。如果'auto'，则表示系数为1/n_features。
- coef0：浮点数，用于指定核函数中的自由项。只有当核函数是'poly'和'sigmoid'有效时才使用它。
- kernel_params：当核函数是个可调用对象时才使用它。如果核函数是上述指定的字符串，则该参数不起作用。
- alpha：一个整数，岭回归的超参数，用于计算逆转换矩阵（当fit_inverse_transform=True时）。
- fit_inverse_transform：一个布尔值。当为True时，需要计算逆转换矩阵。
- eigen_solver：一个字符串，指定求解特征值的算法如下。

- ❍ 'auto'：自动选择。
- ❍ 'dense'：dense特征值求解器。
- ❍ 'arpack'：arpack特征值求解器，用于当特征数量远小于样本数量的情形。

- ❑ tol：一个浮点数，指定arpack特征值求解器的收敛阈值（如果为 0，则自动选择阈值）。
- ❑ max_iter：一个整数，指定arpack特征值求解器的最大迭代次数（如果为None，则自动选择）。
- ❑ remove_zero_eig：一个布尔值。如果为True，则移除所有为零的特征值。如果n_components=None，则也会移除所有为零的特征值。

属性

- ❑ lambdas_：核化矩阵的特征值。
- ❑ alphas_：核化矩阵的特征向量。
- ❑ dual_coef_：逆转换矩阵。

方法

- ❑ fit(X[, y])：训练模型。
- ❑ transform(X)：执行降维。
- ❑ fit_transform(X[, y])：训练模型并且降维。
- ❑ inverse_transform(X)：执行升维（逆向操作），将数据从低维空间逆向转换到原始空间。

首先使用KernelPCA类，给出测试函数：

```
def test_KPCA(*data):
    X,y=data
    kernels=['linear','poly','rbf','sigmoid']
    for kernel in kernels:
        kpca=decomposition.KernelPCA(n_components=None,kernel=kernel)
        kpca.fit(X)
        print('kernel=%s --> lambdas: %s'% (kernel,kpca.lambdas_))
```

然后调用test_KPCA函数：

```
X,y=load_data()
test_KPCA(X,y)
```

运行结果如下。其中因为数据较多，因此这里只是给出了几个较大的 λ，后面的数据用省略号代替。

```
kernel=linear --> lambdas:
[  6.29501274e+02   3.60942922e+01   1.17000623e+01  3.52877104e+00   ...]
kernel=poly --> lambdas:
```

```
[  2.51974731e+05   7.33955085e+03   3.57831449e+03  1.07106819e+03
   1.00406249e+03   1.25469991e+02   5.34250660e+01   1.83918604e+01
   5.86381938e+00   4.72346477e+00   2.39547070e+00   1.83338027e+00
   1.67202391e+00   8.29882720e-01   6.42506753e-01   5.13977885e-01...]
kernel=rbf --> lambdas:
[  4.80818187e+01   1.90919592e+01   6.62368557e+00   4.31294935e+00
   3.73595079e+00   2.24335460e+00   1.76913819e+00   9.42904887e-01
   ...]
kernel=sigmoid --> lambdas:
[  7.05646253e-08   2.50727200e-08   4.24781940e-09  7.45944771e-10   ...]
```

给出绘制降维后样本的分布图的函数：

```
def plot_KPCA(*data):
    X,y=data
    kernels=['linear','poly','rbf','sigmoid']
    fig=plt.figure()
    colors=((1,0,0),(0,1,0),(0,0,1),(0.5,0.5,0),(0,0.5,0.5),(0.5,0,0.5),
        (0.4,0.6,0),(0.6,0.4,0),(0,0.6,0.4),(0.5,0.3,0.2),)

    for i,kernel in enumerate(kernels):
        kpca=decomposition.KernelPCA(n_components=2,kernel=kernel)
        kpca.fit(X)
        X_r=kpca.transform(X)
        ax=fig.add_subplot(2,2,i+1)
        for label ,color in zip( np.unique(y),colors):
            position=y==label
            ax.scatter(X_r[position,0],X_r[position,1],label="target= %d"%label,
            color=color)
        ax.set_xlabel("X[0]")
        ax.set_ylabel("X[1]")
        ax.legend(loc="best")
        ax.set_title("kernel=%s"%kernel)
    plt.suptitle("KPCA")
    plt.show()
```

调用plot_KPCA函数，图形如图 5.2 所示。可以看到，不同的核函数，其降维后的数据分布是不同的。

考察多项式核的参数的影响，给出测试函数：

```
def plot_KPCA_poly(*data):
    X,y=data
    fig=plt.figure()
    colors=((1,0,0),(0,1,0),(0,0,1),(0.5,0.5,0),(0,0.5,0.5),(0.5,0,0.5),
        (0.4,0.6,0),(0.6,0.4,0),(0,0.6,0.4),(0.5,0.3,0.2),)
    Params=[(3,1,1),(3,10,1),(3,1,10),(3,10,10),(10,1,1),(10,10,1),(10,1,10),(10,10,10)]
```

```
    for i,(p,gamma,r) in enumerate(Params):
        kpca=decomposition.KernelPCA(n_components=2,kernel='poly'
        ,gamma=gamma,degree=p,coef0=r)
        kpca.fit(X)
        X_r=kpca.transform(X)
        ax=fig.add_subplot(2,4,i+1)
        for label ,color in zip( np.unique(y),colors):
            position=y==label
            ax.scatter(X_r[position,0],X_r[position,1],label="target= %d"%label,
            color=color)
        ax.set_xlabel("X[0]")
        ax.set_xticks([])
        ax.set_yticks([])
        ax.set_ylabel("X[1]")
        ax.legend(loc="best")
        ax.set_title(r"$ (%s (x \cdot z+1)+%s)^{%s}$"%(gamma,r,p))
    plt.suptitle("KPCA-Poly")
    plt.show()
```

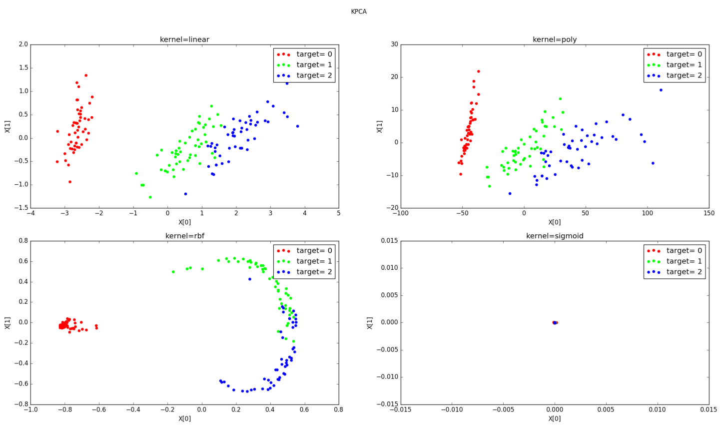


图 5.2 KPCA

调用plot_KPCA_poly函数，图形如图 5.3 所示。可以看到，采用同样的多项式核函数，如果参数不同，其降维后的数据分布是不同的。

考察高斯核的参数的影响，给出测试函数：

```
def plot_KPCA_rbf(*data):
    X,y=data
    fig=plt.figure()
    colors=((1,0,0),(0,1,0),(0,0,1),(0.5,0.5,0),(0,0.5,0.5),(0.5,0,0.5),
        (0.4,0.6,0),(0.6,0.4,0),(0,0.6,0.4),(0.5,0.3,0.2),)
    Gammas=[0.5,1,4,10]
    for i,gamma in enumerate(Gammas):
        kpca=decomposition.KernelPCA(n_components=2,kernel='rbf',gamma=gamma)
```

```
        kpca.fit(X)
        X_r=kpca.transform(X)
        ax=fig.add_subplot(2,2,i+1)
        for label ,color in zip( np.unique(y),colors):
            position=y==label
            ax.scatter(X_r[position,0],X_r[position,1],label="target= %d"%label,
            color=color)
        ax.set_xlabel("X[0]")
        ax.set_xticks([])
        ax.set_yticks([])
        ax.set_ylabel("X[1]")
        ax.legend(loc="best")
        ax.set_title(r"$\exp(-%s||x-z||^2)$"%gamma)
    plt.suptitle("KPCA-rbf")
    plt.show()
```

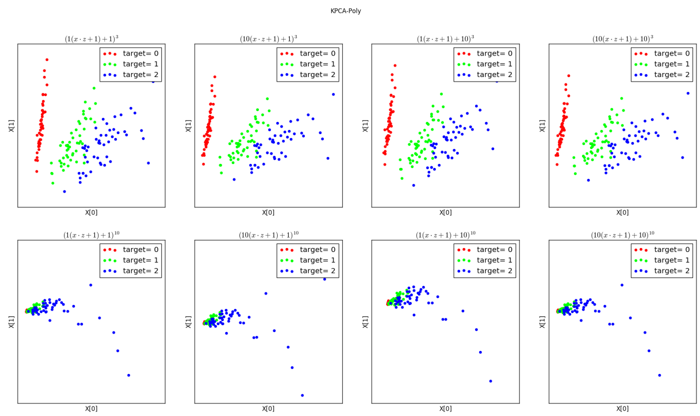


图 5.3 KPCA_poly

调用plot_KPCA_rbf函数，图形如图 5.4 所示。可以看到，采用同样的高斯核函数，如果参数不同，其降维后的数据分布是不同的。

最后考察sigmoid核的参数的影响，给出测试函数：

```
def plot_KPCA_sigmod(*data):
    X,y=data
    fig=plt.figure()
    colors=((1,0,0),(0,1,0),(0,0,1),(0.5,0.5,0),(0,0.5,0.5),(0.5,0,0.5),
        (0.4,0.6,0),(0.6,0.4,0),(0,0.6,0.4),(0.5,0.3,0.2),)
    Params=[(0.01,0.1),(0.01,0.2),(0.1,0.1),(0.1,0.2),(0.2,0.1),(0.2,0.2)]
    for i,(gamma,r) in enumerate(Params):
        kpca=decomposition.KernelPCA(n_components=2,kernel='sigmoid',gamma=gamma,coef0=r)
        kpca.fit(X)
        X_r=kpca.transform(X)
        ax=fig.add_subplot(3,2,i+1)
```

```
        for label ,color in zip( np.unique(y),colors):
            position=y==label
            ax.scatter(X_r[position,0],X_r[position,1],label="target= %d"%label,
            color=color)
        ax.set_xlabel("X[0]")
        ax.set_xticks([])
        ax.set_yticks([])
        ax.set_ylabel("X[1]")
        ax.legend(loc="best")
        ax.set_title(r"$\tanh(%s(x\cdot z)+%s)$"%(gamma,r))
    plt.suptitle("KPCA-sigmoid")
    plt.show()
```

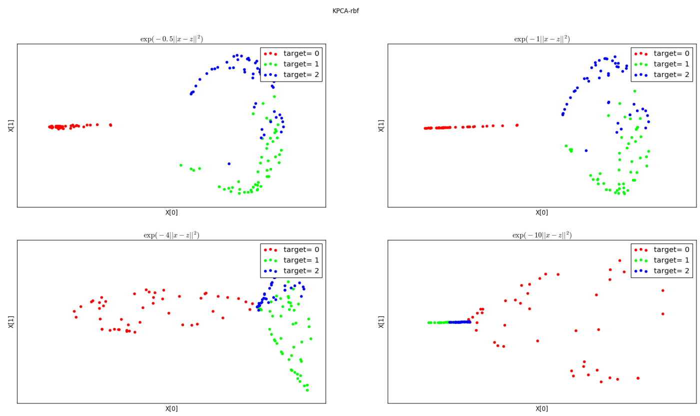


图 5.4 KPCA_rbf

调用plot_KPCA_sigmod函数，图形如图 5.5 所示。可以看到，采用同样的sigmoid核函数，如果参数不同，其降维后的数据分布是不同的。注意这里的 γ 参数的选取要小心。如果 γ 较大，则对于大多数的输入 sigmod函数输出为 1。此时使得核矩阵中大量的位置为 1，从而使得核矩阵不可逆。

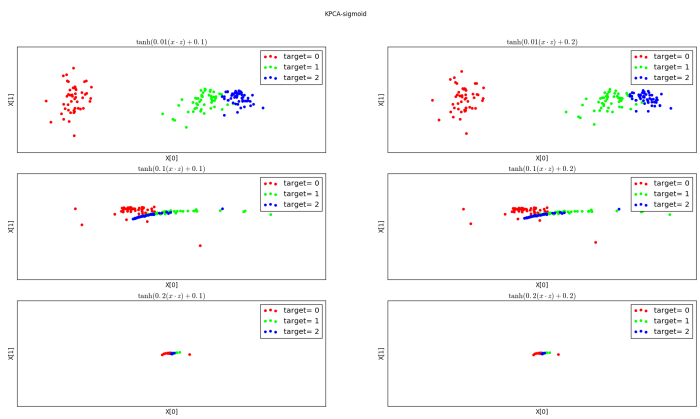


图 5.5 KPCA_sigmod

MDS

MDS是scikit-learn实现的多维缩放模型，其原型为：

```
class sklearn.manifold.MDS(n_components=2, metric=True, n_init=4, max_iter=300,
 verbose=0, eps=0.001, n_jobs=1, random_state=None, dissimilarity='euclidean')
```

参数

- metric：一个布尔值。如果为True，则使用距离度量；否则使用非距离度量SMACOF。
- n_components：一个整数，指定低维空间。
- n_init：一个整数，指定初始化的次数。在使用SMACOF算法时，会选择n_init次不同的初始值，然后选择这些结果中最好的那个作为最终结果。
- max_iter：一个整数。指定在使用SMACOF算法时，得到一轮结果需要的最大迭代次数。
- eps：一个浮点数，用于指定收敛阈值。
- n_jobs：一个整数，指定并行性。默认为 −1 表示派发任务到所有计算机的 CPU 上。
- random_state：一个整数或者一个RandomState实例，或者None。
  - 如果为整数，则它指定了随机数生成器的种子。
  - 如果为RandomState实例，则指定了随机数生成器。
  - 如果为None，则使用默认的随机数生成器。
- dissimilarity: 一个字符串值，用于定义如何计算不相似度。可以为如下。
  - 'euclidean'：使用欧氏距离。
  - 'precomputed'：由使用者提供距离矩阵。

属性

- embedding_：给出了原始数据集在低维空间中的嵌入矩阵。
- stress_：一个浮点数，给出了不一致的距离的总和。

方法

- fit(X[, y, init])：训练模型。
- fit_transform(X[, y, init])：训练模型并返回转换后的低维坐标。

首先使用MDS类，给出测试函数：

```
def test_MDS(*data):
    X,y=data
    for n in [4,3,2,1]:
        mds=manifold.MDS(n_components=n)
        mds.fit(X)
        print('stress(n_components=%d) : %s'% (n, str(mds.stress_)))
```

然后调用test_MDS函数：

```
X,y=load_data()
test_MDS(X,y)
```

运行结果如下：

```
stress(n_components=4) : 12.0577408711
stress(n_components=3) : 17.8262808779
stress(n_components=2) : 234.395807108
stress(n_components=1) : 23691.9560412
```

原始数据的特征为四维的，可以看到将原始数据嵌入到四维空间时，距离误差之和为 12.0577（在原空间中嵌入，理论上不存在误差。但是由于计算精度的影响，这里有一定误差）。将原始数据嵌入到三维空间时，距离误差之和为 17.8263（与原始空间的结果相差较小）。而将原始数据嵌入到二维和一维时，距离的误差之和分别为 234.4 和 23692，与原始空间的结果相差较大。

> 该指标并不能用于判定降维效果的好坏，它只是一个中性指标。

给出绘制降维后的样本的分布图的函数：

```
def plot_MDS(*data):
    X,y=data
    mds=manifold.MDS(n_components=2)
    X_r=mds.fit_transform(X)

    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    colors=((1,0,0),(0,1,0),(0,0,1),(0.5,0.5,0),(0,0.5,0.5),(0.5,0,0.5),
        (0.4,0.6,0),(0.6,0.4,0),(0,0.6,0.4),(0.5,0.3,0.2),)
    for label ,color in zip( np.unique(y),colors):
        position=y==label
        ax.scatter(X_r[position,0],X_r[position,1],label="target= %d"%label,color=color)

    ax.set_xlabel("X[0]")
    ax.set_ylabel("X[1]")
    ax.legend(loc="best")
    ax.set_title("MDS")
    plt.show()
```

调用plot_MDS函数，图形如图 5.6 所示。

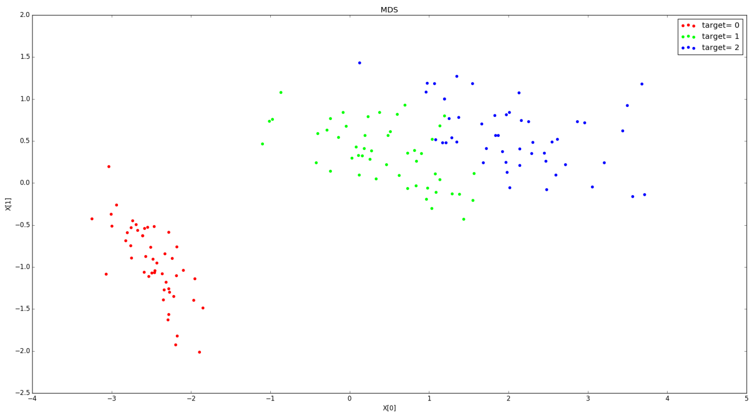


图 5.6 MDS

Isomap

Isomap类是scikit-learn提供的Isomap模型，其原型为：

```
class sklearn.manifold.Isomap(n_neighbors=5, n_components=2, eigen_solver='auto',
 tol=0, max_iter=None, path_method='auto', neighbors_algorithm='auto')
```

参数

- n_neighbors：一个整数，指定近邻参数 k。
- n_components：一个整数，指定低维的维数。
- eigen_solver：一个字符串，指定求解特征值的算法，可以为如下。
  - 'auto'：由算法自动选取。
  - 'arpack'：使用 Arnoldi分解算法。
  - 'dense'：使用一个直接求解特征值的算法（如LAPACK）。
- tol：一个浮点数，指定求解特征值算法的收敛阈值（当eigen_solver='dense'时，该参数无用）。
- max_iter：一个浮点数，指定求解特征值算法的最大迭代次数（当eigen_solver='dense'时，该参数无用）。
- path_method：一个字符串，指定寻找最短路径算法。可以为如下。
  - 'auto'：由算法自动选取。
  - 'FW'：使用Floyd_Warshall算法。
  - 'D'：使用Dijkstra算法。
- neighbors_algorithm：一字符串，指定计算最近邻的算法。可以为如下。
  - 'ball_tree'：使用 BallTree算法。
  - 'kd_tree：使用 KDTree算法。
  - 'brute'：使用暴力搜索法。

❍ 'auto'：自动决定最合适的算法。

属性

- embedding_：给出了原始数据集在低维空间中的嵌入矩阵。
- training_data_：存储了原始训练数据。
- dist_matrix_：存储了原始训练数据的距离矩阵。

方法

- fit(X[, y])：训练模型。
- transform(X)：转换X到低维空间。
- fit_transform(X[, y])：训练模型并将原始数据集转换到低维空间。
- reconstruction_error()：计算重构误差。

首先使用Isomap类，给出测试函数：

```
def test_Isomap(*data):
    X,y=data
    for n in [4,3,2,1]:
        isomap=manifold.Isomap(n_components=n)
        isomap.fit(X)
        print('reconstruction_error(n_components=%d) : %s'%
            (n, isomap.reconstruction_error()))
```

然后调用test_Isomap函数：

```
X,y=load_data()
test_Isomap(X,y)
```

运行结果如下：

```
reconstruction_error(n_components=4) : 1.00971800681
reconstruction_error(n_components=3) : 1.01828451463
reconstruction_error(n_components=2) : 1.02769837643
reconstruction_error(n_components=1) : 1.07166427632
```

可以看到对于不同的低维空间，其降维的重构误差比较小。

> 该指标并不能用于判定降维效果的好坏，它只是一个中性指标。

给出绘制降维后的样本的分布图的函数：

```
def plot_Isomap_k(*data):
    X,y=data
    Ks=[1,5,25,y.size-1]
```

```
    fig=plt.figure()
    for i, k in enumerate(Ks):
        isomap=manifold.Isomap(n_components=2,n_neighbors=k)
        X_r=isomap.fit_transform(X)

        ax=fig.add_subplot(2,2,i+1)
        colors=((1,0,0),(0,1,0),(0,0,1),(0.5,0.5,0),(0,0.5,0.5),(0.5,0,0.5),
            (0.4,0.6,0),(0.6,0.4,0),(0,0.6,0.4),(0.5,0.3,0.2),)
        for label ,color in zip( np.unique(y),colors):
            position=y==label
            ax.scatter(X_r[position,0],X_r[position,1],label="target= %d"
            %label,color=color)

        ax.set_xlabel("X[0]")
        ax.set_ylabel("X[1]")
        ax.legend(loc="best")
        ax.set_title("k=%d"%k)
    plt.suptitle("Isomap")
    plt.show()
```

调用plot_Isomap_k函数，图形如图 5.7 所示。可以看到，$k = 1$ 时，近邻范围过小，此时发生断路现象。本应该相连的区域限制被认定为不相连。

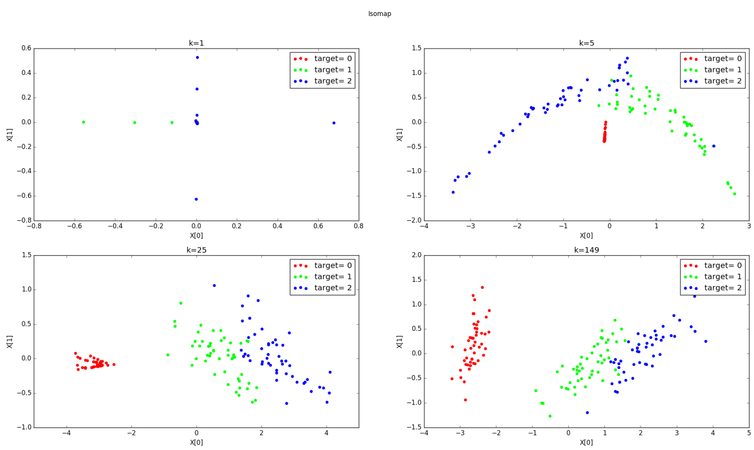


图 5.7 Isomap_k

最后给出将原始数据的特征直接压缩到一维时的情形。给出测试函数：

```
def plot_Isomap_k_d1(*data):
    X,y=data
    Ks=[1,5,25,y.size-1]

    fig=plt.figure()
    for i, k in enumerate(Ks):
```

```
        isomap=manifold.Isomap(n_components=1,n_neighbors=k)
        X_r=isomap.fit_transform(X)

        ax=fig.add_subplot(2,2,i+1)
        colors=((1,0,0),(0,1,0),(0,0,1),(0.5,0.5,0),(0,0.5,0.5),(0.5,0,0.5),
            (0.4,0.6,0),(0.6,0.4,0),(0,0.6,0.4),(0.5,0.3,0.2),)
        for label ,color in zip( np.unique(y),colors):
            position=y==label
            ax.scatter(X_r[position],np.zeros_like(X_r[position]),
            label="target= %d"%label,color=color)

        ax.set_xlabel("X")
        ax.set_ylabel("Y")
        ax.legend(loc="best")
        ax.set_title("k=%d"%k)
    plt.suptitle("Isomap")
    plt.show()
```

调用plot_Isomap_k函数，图形如图 5.8 所示。

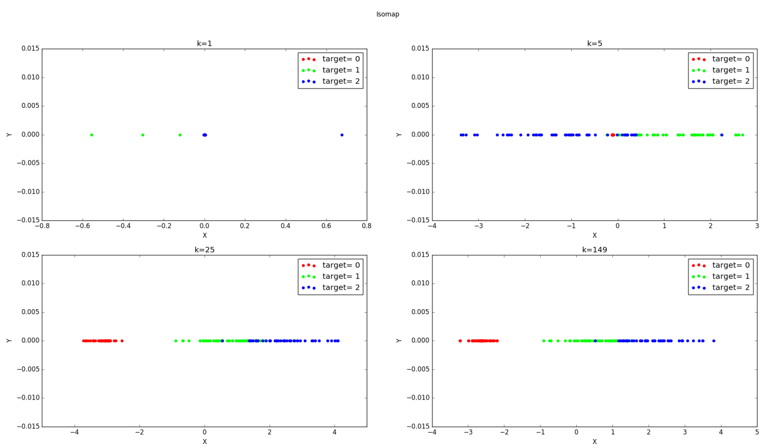


图 5.8 Isomap_k_d1

LLE 模型

LocallyLinearEmbedding是 scikit-learn提供的LLE模型，其原型为：

```
class sklearn.manifold.LocallyLinearEmbedding(n_neighbors=5, n_components=2, reg=0.001,
 eigen_solver='auto', tol=1e-06, max_iter=100, method='standard', hessian_tol=0.0001,
 modified_tol=1e-12, neighbors_algorithm='auto', random_state=None)
```

参数

- ❑ n_neighbors：一个整数，指定近邻参数 k。

- n_components：一个整数，指定低维的维数。
- reg：一个浮点数，正则化项的系数。
- eigen_solver：一个字符串，指定求解特征值的算法，可以为如下。
  - 'auto'：由算法自动选取。
  - 'arpack'：使用 Arnoldi分解算法。
  - 'dense'：使用一个直接求解特征值的算法（如LAPACK）。
- tol：一个浮点数，指定求解特征值算法的收敛阈值（当eigen_solver='dense'时，该参数无用）。
- max_iter：一个浮点数，指定求解特征值算法的最大迭代次数（当 eigen_solver='dense'时，该参数无用）。
- method：一个字符串，用于指定LLE算法的形式。可以为如下。
  - 'standard'：使用标准的LLE算法。
  - 'hessian'：使用Hessian eignmap算法。
  - 'modified'：使用modified LLE算法。
  - 'ltsa'：使用local tangent space alignment算法。
- hessian_tol：一个浮点数，用于method='hessian'时收敛的阈值。
- modified_tol：一个浮点数，用于method='modified'时收敛的阈值。
- neighbors_algorithm：一字符串，指定计算最近邻的算法。可以为如下。
  - 'ball_tree'：使用 BallTree算法。
  - 'kd_tree：使用 KDTree算法。
  - 'brute'：使用暴力搜索法。
  - 'auto'：自动决定最合适的算法。
- random_state：一个整数或者一个RandomState实例，或者None，用于eigen_solver='arpack'。
  - 如果为整数，则它指定了随机数生成器的种子。
  - 如果为RandomState实例，则指定了随机数生成器。
  - 如果为None，则使用默认的随机数生成器。

属性

- embedding_vectors_：给出了原始数据在低维空间的嵌入矩阵。
- reconstruction_error_：给出了重构误差。

方法

- fit(X[, y])：训练模型。
- transform(X)：将X转换到低维空间。
- fit_transform(X[, y])：训练模型并将原始数据集转换到低维空间。

首先使用LocallyLinearEmbedding类，给出测试函数：

```
def test_LocallyLinearEmbedding(*data):
    X,y=data
    for n in [4,3,2,1]:
        lle=manifold.LocallyLinearEmbedding(n_components=n)
        lle.fit(X)
        print('reconstruction_error(n_components=%d) : %s'%
            (n, lle.reconstruction_error_))
```

然后调用test_LocallyLinearEmbedding函数：

```
X,y=load_data()
test_LocallyLinearEmbedding(X,y)
```

运行结果如下：

```
reconstruction_error(n_components=4) : 7.19936880176e-07
reconstruction_error(n_components=3) : 3.8706050149e-07
reconstruction_error(n_components=2) : 6.64141991211e-08
reconstruction_error(n_components=1) : -1.74047846991e-15
```

可以看到对于不同的低维空间，其降维的重构误差非常小。

> 该指标并不能用于判定降维效果的好坏，它只是一个中性指标。

给出绘制降维后的样本的分布图的函数：

```
def plot_LocallyLinearEmbedding_k(*data):
    X,y=data
    Ks=[1,5,25,y.size-1]

    fig=plt.figure()
    for i, k in enumerate(Ks):
        lle=manifold.LocallyLinearEmbedding(n_components=2,n_neighbors=k)
        X_r=lle.fit_transform(X)

        ax=fig.add_subplot(2,2,i+1)
        colors=((1,0,0),(0,1,0),(0,0,1),(0.5,0.5,0),(0,0.5,0.5),(0.5,0,0.5),
            (0.4,0.6,0),(0.6,0.4,0),(0,0.6,0.4),(0.5,0.3,0.2),)
        for label ,color in zip( np.unique(y),colors):
            position=y==label
            ax.scatter(X_r[position,0],X_r[position,1],label="target= %d"
            %label,color=color)

        ax.set_xlabel("X[0]")
```

```
        ax.set_ylabel("X[1]")
        ax.legend(loc="best")
        ax.set_title("k=%d"%k)
    plt.suptitle("LocallyLinearEmbedding")
    plt.show()
```

调用plot_LocallyLinearEmbedding_k函数，图形如图 5.9 所示。可以看到，$k = 1, 5$ 时，近邻范围过小，同样发生了断路现象。

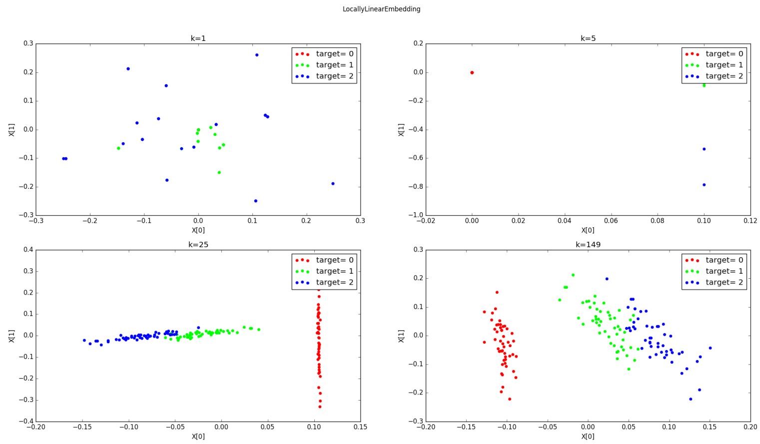


图 5.9 LocallyLinearEmbedding_k

最后给出将原始数据的特征直接压缩到一维时的情形。给出测试函数：

```
def plot_LocallyLinearEmbedding_k_d1(*data):
    X,y=data
    Ks=[1,5,25,y.size-1]

    fig=plt.figure()
    for i, k in enumerate(Ks):
        lle=manifold.LocallyLinearEmbedding(n_components=1,n_neighbors=k)
        X_r=lle.fit_transform(X)

        ax=fig.add_subplot(2,2,i+1)
        colors=((1,0,0),(0,1,0),(0,0,1),(0.5,0.5,0),(0,0.5,0.5),(0.5,0,0.5),
            (0.4,0.6,0),(0.6,0.4,0),(0,0.6,0.4),(0.5,0.3,0.2),)
        for label ,color in zip( np.unique(y),colors):
            position=y==label
            ax.scatter(X_r[position],np.zeros_like(X_r[position]),
            label="target= %d"%label,color=color)

        ax.set_xlabel("X")
        ax.set_ylabel("Y")
        ax.legend(loc="best")
        ax.set_title("k=%d"%k)
```

```
plt.suptitle("LocallyLinearEmbedding")
plt.show()
```

调用plot_LocallyLinearEmbedding_k_d1函数，图形如图 5.10 所示。

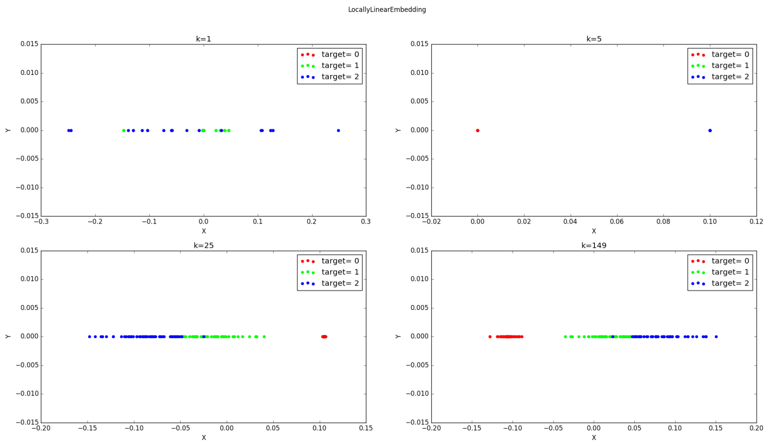


图 5.10 LocallyLinearEmbedding_k_d1

5.4 小结

对原始数据采取降维的原因通常有两个：缓解“维度灾难”或者对数据进行可视化。PCA主成分分析是一种著名的无监督的线性降维方法，LDA线性判别分析是一种著名的监督线性降维方法。而流形学习是借鉴拓扑流形概念的降维方法。

降维的效果好坏没有一个直接的指标。通常通过对数据进行降维，然后用降维后的数据进行学习，再根据学习的效果选择一个恰当的降维方式和一个合适的降维模型参数。

scikit-learn中提供了一些经验技巧：

- 如果对某个维度的特征进行了缩放，则确保对所有的维度的特征也进行同样处理。因为流形学习是基于距离来计算的。当改变了某个维度的数据的倍数时（比如修改了量纲，将 m修改成cm），会直接影响流形学习的结果。
- 可以利用重建误差来选择一个合适的低维空间。在流形学习中，随着维度递增到n_components，重建误差会递减。
- 流形学习对于噪声数据非常敏感。噪声数据可能出现在两个区域的连接处：如果没有出现噪声，这两个区域是断路的；如果出现噪声，这两个区域是短路的。

第 6 章

聚类和 EM 算法

6.1 概述

当训练样本的标记信息未知时，此时称为无监督学习（unsupervised learning）。无监督学习通过对无标记训练样本的学习来寻找这些数据的内在性质，其主要工具是聚类（clustering）算法。

聚类的思想是：将数据集划分为若干个不相交子集（称为一个簇cluster），每个簇潜在地对应于某一个概念。但是聚类过程仅仅能生成簇结构，而每个簇所代表的概念的语义由使用者自己解释。也就是聚类算法并不会告诉你：它生成的这些簇分别代表什么意义。它只会告诉你：算法已经将数据集划分为这些不相交的簇了。

用数学语言描述聚类：给定样本集 $D=\{\vec{\mathbf{x}}_1,\vec{\mathbf{x}}_2,\cdots,\vec{\mathbf{x}}_N\}$（假设样本集包含 N 个无标记样本）。样本 $\vec{\mathbf{x}}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(n)})^T\in\mathbb{R}^n$。聚类算法将样本集 D 划分成 K 个不相交的簇 $\{C_1,C_2,\cdots,C_K\}$，其中 $C_k\bigcap_{k\neq l}C_l=\phi,\quad D=\bigcup_{k=1}^K C_k$。

令 $\lambda_i\in\{1,2,\cdots,K\}$ 表示样本 $\vec{\mathbf{x}}_i$ 的簇标记cluster label，即 $\vec{\mathbf{x}}_i\in C_{\lambda_i}$，则聚类的结果可以用簇标记向量 $\vec{\lambda}=(\lambda_1,\lambda_2,\cdots,\lambda_N)^T$ 来表示。

聚类的作用：

- 可以作为一个单独的过程，用于寻找数据的一个分布规律；
- 也可以作为分类的预处理过程。首先对分类数据进行聚类处理，然后在聚类结果的每一个簇上执行分类过程。

6.2　算法笔记精华

6.2.1　聚类的有效性指标

聚类的有效性指标（validity index）用于度量聚类的效果。很显然，希望彼此相似的样本尽量在同一簇，彼此不相似的样本尽量在不同的簇。也就是：同一簇的样本尽可能地彼此相似，不同簇的样本之间尽可能地不同。

聚类的性能度量分为以下两类。

- 外部指标external index：该指标是由聚类结果与某个参考模型reference model进行比较而获得的。
- 内部指标internal index：该指标直接由考察聚类结果而得到，并不利用任何参考模型。

外部指标

给定数据集 $D=\{\vec{x}_1,\vec{x}_2,\cdots,\vec{x}_N\}$ 。假定某个参考模型给出的簇划分为 $\mathcal{C}^*=\{C_1^*,C_2^*,\cdots,C_{K'}^*\}$，其簇标记向量为 $\vec{\lambda}^*$。若聚类算法给出的簇划分为 $\mathcal{C}=\{C_1,C_2,\cdots,C_K\}$，其簇标记向量为 $\vec{\lambda}$ 。定义：

$$a=|SS|,SS=\{(\vec{x}_i,\vec{x}_j)\mid\lambda_i=\lambda_j,\lambda_i^*=\lambda_j^*,i<j\}$$
$$b=|SD|,SD=\{(\vec{x}_i,\vec{x}_j)\mid\lambda_i=\lambda_j,\lambda_i^*\neq\lambda_j^*,i<j\}$$
$$c=|DS|,DS=\{(\vec{x}_i,\vec{x}_j)\mid\lambda_i\neq\lambda_j,\lambda_i^*=\lambda_j^*,i<j\}$$
$$d=|DD|,DD=\{(\vec{x}_i,\vec{x}_j)\mid\lambda_i\neq\lambda_j,\lambda_i^*\neq\lambda_j^*,i<j\}$$

其中，$|\cdot|$ 表示集合的元素的个数。各集合的意义如下。

- SS：包含了同时隶属于 $\mathcal{C},\mathcal{C}^*$ 的样本对。
- SD：包含了隶属于 $\mathcal{C}$ ，但是不隶属于 $\mathcal{C}^*$ 的样本对。
- DS：包含了不隶属于 $\mathcal{C}$，但是隶属于 $\mathcal{C}^*$ 的样本对。
- DD：包含了既不隶属于 $\mathcal{C}$，又不隶属于 $\mathcal{C}^*$ 的样本对。

由于每个样本对 $(\vec{x}_i,\vec{x}_j),i<j$ 仅能出现在一个集合中，因此有 $a+b+c+d=\frac{N(N-1)}{2}$。

使用上述定义式，可以有下面这些外部指标。

- Jaccard系数Jaccard Coefficient:JC：

$$JC=\frac{a}{a+b+c}$$

  它刻画了所有属于同一类的样本对（要么在 $\mathcal{C}$ 中属于同一类，要么在 $\mathcal{C}^*$ 中属于同一类），同时在 $\mathcal{C},\mathcal{C}^*$ 中隶属于同一类的样本对的比例。

- FM指数Fowlkes and Mallows Index:FMI：

$$FMI = \sqrt{\frac{a}{a+b} \cdot \frac{a}{a+c}}$$

  它刻画的是：在 $\mathcal{C}$ 中属于同一类的样本对中，同时属于 $\mathcal{C}^*$ 的样本对的比例为 $p1$ ；在 $\mathcal{C}^*$ 中属于同一类的样本对中，同时属于 $\mathcal{C}$ 的样本对的比例为 $p2$，FMI就是 $p1$ 和 $p2$ 的几何平均。

- Rand指数（Rand Index，RI）：

$$RI = \frac{2(a+d)}{N(N-1)}$$

  它刻画的是同时隶属于 $\mathcal{C},\mathcal{C}^*$ 的样本对与既不隶属于 $\mathcal{C}$，又不隶属于 $\mathcal{C}^*$ 的样本对之和占所有样本对的比例。

- ARI指数（Adjusted Rand Index）：

$$ARI = \frac{RI - E[RI]}{\max(RI) - E[RI]}$$

  使用RI时有个问题，就是对于随机聚类，RI指数不保证接近0（可能还很大）。而ARI指数就可通过利用随机聚类情况下的RI（即E[RI]）来解决这个问题。

这些外部指标性能度量的结果都在[0,1]之间。这些值越大，说明聚类的性能越好。

内部指标

给定数据集 $D = \{\vec{x}_1, \vec{x}_2, \cdots, \vec{x}_N\}$ 。若聚类给出的簇划分为 $\mathcal{C} = \{C_1, C_2, \cdots, C_K\}$，定义：

$$\text{avg}(C_k) = \frac{2}{|C_k|(|C_k|-1)} \sum_{\vec{x}_i,\vec{x}_j \in C_k, i \neq j} \text{distance}(\vec{x}_i, \vec{x}_j), \quad k = 1,2,\cdots,K$$

$$\text{diam}(C_k) = \max_{\vec{x}_i,\vec{x}_j \in C_k, i \neq j} \text{distance}(\vec{x}_i, \vec{x}_j), \quad k = 1,2,\cdots,K$$

$$d_{min}(C_k, C_l) = \min_{\vec{x}_i \in C_k, \vec{x}_j \in C_l} \text{distance}(\vec{x}_i, \vec{x}_j), \quad k,l = 1,2,\cdots,K; k \neq l$$

$$d_{cen}(C_k, C_l) = \text{distance}(\vec{\mu}_k, \vec{\mu}_l), \quad k,l = 1,2,\cdots,K; k \neq l$$

其中，$\text{distance}(\vec{x}_i, \vec{x}_j)$ 表示两点 $\vec{x}_i, \vec{x}_j$ 之间的距离；$\vec{\mu}_k$ 表示簇 C_k 的中心点，$\vec{\mu}_l$ 表示簇 C_l 的中心点；$\text{distance}(\vec{\mu}_k, \vec{\mu}_l)$ 表示簇 C_k, C_l 的中心点之间的距离。上述定义的意义如下。

- $\text{avg}(C_k)$：簇 C_k 中每对样本之间的平均距离。
- $\text{diam}(C_k)$：簇 C_k 中距离最远的两个点的距离。
- $d_{min}(C_k, C_l)$：簇 C_k, C_l 之间最近的距离。
- $d_{cen}(C_k, C_l)$：簇 C_k, C_l 中心点之间的距离。

使用上述定义式，可有以下的内部指标。

- DB指数（Davies-Bouldin Index，DBI）：

$$DBI = \frac{1}{K}\sum_{k=1}^{K}\max_{k\neq l}\left(\frac{\text{avg}(C_k)+\text{avg}(C_l)}{d_{cen}(C_k,C_l)}\right)$$

  它刻画的是：给定两个簇，每个簇样本之间平均值之和比上两个簇的中心点之间的距离作为度量。然后考察该度量对所有簇的平均值。显然 DBI 越小越好。如果每个簇样本之间的平均值越小（即簇内样本距离都很近），则 DBI 越小；如果簇间中心点的距离越大（即簇间样本距离相互都很远），则 DBI 越小。

- Dunn指数（Dunn Index，DI）：

$$DI = \frac{\min_{k\neq l} d_{min}(C_k,C_l)}{\max_i \text{diam}(C_i)}$$

  它刻画的是：任意两个簇之间最近的距离的最小值，除以任意一个簇内距离最远的两个点的距离的最大值。DI 越大越好。如果任意两个簇之间最近的距离的最小值越大（即簇间样本距离相互都很远），则 DI 越大；如果任意一个簇内距离最远的两个点的距离的最大值越小（即簇内样本距离都很近），则 DI 越大。

6.2.2 距离度量

常用的距离函数 distance$(\cdot,\cdot)$ 有以下两种距离。

- 闵可夫斯基距离（Minkowski distance）：给定样本 $\vec{\boldsymbol{x}}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(n)})^T,\vec{\boldsymbol{x}}_j=(x_j^{(1)},x_j^{(2)},\cdots,x_j^{(n)})^T$，则闵可夫斯基距离定义为：

$$\text{distance}(\vec{\boldsymbol{x}}_i,\vec{\boldsymbol{x}}_j)=\left(\sum_{d=1}^{n}|x_i^{(d)}-x_j^{(d)}|^p\right)^{1/p}$$

  - 当 $p=2$ 时，闵可夫斯基距离就是欧式距离（Euclidean distance）：

$$\text{distance}(\vec{\boldsymbol{x}}_i,\vec{\boldsymbol{x}}_j)=||\vec{\boldsymbol{x}}_i-\vec{\boldsymbol{x}}_j||_2=\sqrt{\sum_{d=1}^{n}|x_i^{(d)}-x_j^{(d)}|^2}$$

  - 当 $p=1$ 时，闵可夫斯基距离就是曼哈顿距离（Manhattan distance）：

$$\text{distance}(\vec{\boldsymbol{x}}_i,\vec{\boldsymbol{x}}_j)=||\vec{\boldsymbol{x}}_i-\vec{\boldsymbol{x}}_j||_1=\sum_{d=1}^{n}|x_i^{(d)}-x_j^{(d)}|$$

- ❑ VDM距离（Value Difference Metric）：考虑非数值类属性（如属性取值为：中国，印度，美国，英国），令 $m_{d,a}$ 表示 $x^{(d)}=a$ 的样本数；$m_{d,a,k}$ 表示 $x^{(d)}=a$ 且位于簇 C_k 中的样本的数量。则在属性 d 上的两个取值 a,b 之间的VDM距离为：

$$VDM_p(a,b)=\left(\sum_{k=1}^{K}|\frac{m_{d,a,k}}{m_{d,a}}-\frac{m_{d,b,k}}{m_{d,b}}|^p\right)^{1/p}$$

VDM距离刻画的是属性取值在各簇上的频率分布之间的差异。

当样本的属性为数值属性与非数值属性混合时，可以将闵可夫斯基距离与 VDM 距离混合使用。假设属性 $x^{(1)},x^{(2)},\cdots,x^{(n_c)}$ 为数值属性，属性 $x^{(n_c+1)},x^{(n_c+2)},\cdots,x^{(n)}$ 为非数值属性。则：

$$\text{distance}(\vec{\boldsymbol{x}}_i,\vec{\boldsymbol{x}}_j)=\left(\sum_{d=1}^{n_c}|x_i^{(d)}-x_j^{(d)}|^p+\sum_{d=n_c+1}^{n}VDM_p(x_i^{(d)},x_j^{(d)})\right)^{1/p}$$

对于样本空间中不同属性的重要性不同的情况，此时可以使用加权距离。如加权闵可夫斯基距离为：

$$\text{distance}(\vec{\boldsymbol{x}}_i,\vec{\boldsymbol{x}}_j)=\left(\sum_{d=1}^{n}w_d|x_i^{(d)}-x_j^{(d)}|^p\right)^{1/p}$$

$$w_d\geqslant 0,d=1,2,\cdots,n$$

$$\sum_{d=1}^{n}w_d=1$$

注意：这里的距离度量满足三角不等式：$\text{distance}(\vec{\boldsymbol{x}}_i,\vec{\boldsymbol{x}}_j)\leqslant\text{distance}(\vec{\boldsymbol{x}}_i,\vec{\boldsymbol{x}}_k)+\text{distance}(\vec{\boldsymbol{x}}_k,\vec{\boldsymbol{x}}_j)$。但是在某些任务中，需要采用非度量距离non-metric distance（即不满足这一三角不等式的距离）。如：美人鱼和人距离很近，美人鱼和鱼距离很近，但是人和鱼的距离很远。

6.2.3 原型聚类

原型聚类（prototype-based clustering）是假设聚类结构能通过一组原型刻画。常用的原型聚类有：

- ❑ k均值算法k-means。
- ❑ 高斯混合聚类Mixture-of-Gaussian。

k 均值算法

给定样本集 $D=\{\vec{\boldsymbol{x}}_1,\vec{\boldsymbol{x}}_2,\cdots,\vec{\boldsymbol{x}}_N\}$，假设聚类的簇划分 $\mathcal{C}=\{C_1,C_2,\cdots,C_K\}$。k 均值算法的目标是最小化平方误差：

$$err=\sum_{k=1}^{K}\sum_{\vec{\boldsymbol{x}}_i\in C_k}||\vec{\boldsymbol{x}}_i-\vec{\mu}_k||_2^2$$

其中，$\vec{\mu}_k=\frac{1}{|C_k|}\sum_{\vec{\boldsymbol{x}}_i\in C_k}\vec{\boldsymbol{x}}_i$ 是簇 C_k 的均值向量。上式刻画了簇类样本围绕簇均值向量的紧密程度，err 越小，则簇内样本距簇均值向量越紧密。

现在要求最优化问题：

$$\min_{\mathcal{C}}\sum_{k=1}^{K}\sum_{\vec{\boldsymbol{x}}_i\in C_k}||\vec{\boldsymbol{x}}_i-\vec{\mu}_k||_2^2$$

该问题的求解需要考察样本集合 D 的所有可能的簇划分。k均值算法采用贪心策略，通过迭代优化来近似求解。其原理为：k均值算法首先假设一组向量作为所有簇的簇均值向量，然后根据假设的簇均值向量给出了数据集 D 的一个簇划分，再根据这个簇划分来计算真实的簇均值向量。

- 如果真实的簇均值向量等于假设的簇均值向量，则说明假设正确。根据假设簇均值向量给出的数据集 D 的一个簇划分确实是原问题的解。
- 如果真实的簇均值向量不等于假设的簇均值向量，则可以将真实的簇均值向量作为新的假设簇均值向量，继续求解。

这里的一个关键就是：给定假设的簇均值向量，如何计算出数据集 D 的一个簇划分？k均值算法的策略是：样本离哪个簇的簇均值向量近，则该样本就划归到那个簇。

下面给出k均值算法。

- 输入：
  - 样本集 $D=\{\vec{\boldsymbol{x}}_1,\vec{\boldsymbol{x}}_2,\cdots,\vec{\boldsymbol{x}}_N\}$。
  - 聚类簇数 K。
- 输出：簇划分 $\mathcal{C}=\{C_1,C_2,\cdots,C_K\}$。
- 算法步骤：
  - 从 D 中随机选择 K 个样本作为初始簇均值向量 $\{\vec{\mu}_1,\vec{\mu}_2,\cdots,\vec{\mu}_K\}$。
  - 重复迭代直到算法收敛，迭代内容如下。
    - 初始化阶段：取 $C_k=\phi,k=1,2,\cdots,K$。

      > 这是因为，要计算每一轮迭代中每个簇实际的簇均值向量。

    - 划分阶段：令 $i=1,2,\cdots,N$
      - ★ 计算 $\vec{\boldsymbol{x}}_i$ 的簇标记如下。

$$\lambda_i=\arg\min_k||\vec{\boldsymbol{x}}_i-\vec{\mu}_k||_2,k\in\{1,2,\cdots,K\}$$

> 即：如果 $\vec{\boldsymbol{x}}_i$ 里哪个簇的簇均值向量最近，则该样本就标记为那个簇。

- ★ 然后将样本 $\vec{\boldsymbol{x}}_i$ 划入相应的簇：

$$C_{\lambda_i} = C_{\lambda_i} \bigcup \{\vec{\boldsymbol{x}}_i\}$$

❖ 重计算阶段：计算 $\hat{\vec{\mu}}_k$

$$\hat{\vec{\mu}}_k = \frac{1}{|C_k|} \sum_{\vec{\boldsymbol{x}}_i \in C_k} \vec{\boldsymbol{x}}_i$$

❖ 终止条件判断：

- ★ 如果对所有的 $k \in \{1, 2, \cdots, K\}$，都有 $\hat{\vec{\mu}}_k = \vec{\mu}_k$，则算法收敛，终止迭代。
- ★ 否则重赋值 $\vec{\mu}_k = \hat{\vec{\mu}}_k$。

高斯混合聚类

高斯混合聚类通过概率模型来表示聚类原型。对服从高斯分布的 n 维随机向量 $\vec{\boldsymbol{x}}$ 来说，其概率密度函数为：

$$p(\vec{\boldsymbol{x}} \mid \vec{\mu}, \Sigma) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\vec{\boldsymbol{x}} - \vec{\mu})^T \Sigma^{-1} (\vec{\boldsymbol{x}} - \vec{\mu})\right)$$

其中 $\vec{\mu} = (\mu^{(1)}, \mu^{(2)}, \cdots, \mu^{(n)})^T$ 为 n 维均值向量，Σ 是 $n \times n$ 的协方差矩阵。$p(\vec{\boldsymbol{x}} \mid \vec{\mu}, \Sigma)$ 表示 $\vec{\boldsymbol{x}}$ 的概率密度函数由参数 $\vec{\mu}, \Sigma$ 决定。

高斯混合分布定义如下：

$$p_{\mathcal{M}}(\vec{\boldsymbol{x}}) = \sum_{k=1}^{K} \alpha_k p(\vec{\boldsymbol{x}} \mid \vec{\mu}_k, \Sigma_k)$$

高斯混合分布由 K 个成分混合而成，其中每个成分对应一个高斯分布。$\vec{\mu}_k, \Sigma_k$ 是第 k 个成分对应的高斯分布的参数。$\alpha_k > 0$ 是第 k 个成分的混合系数，且 $\sum_{k=1}^{K} \alpha_k = 1$。

假设样本训练集 $D = \{\vec{\boldsymbol{x}}_1, \vec{\boldsymbol{x}}_2, \cdots, \vec{\boldsymbol{x}}_N\}$ 的生成过程是由高斯混合分布给出的。令随机变量 $Z \in \{1, 2, \cdots, K\}$，那么 Z 的概率分布 $p(Z = k) = \alpha_k$，记为 $p_\alpha(k)$。生成样本的过程分为两步：

- ❑ 首先根据概率分布 $p_\alpha(k)$ 生成随机变量 Z。
- ❑ 再根据 Z 的结果，比如 $Z = 2$，根据概率 $p(\vec{\boldsymbol{x}} \mid \vec{\mu}_2, \Sigma_2)$ 生成样本。

根据贝叶斯定理，若已知输出为 $\vec{\boldsymbol{x}}_i$，则 Z 的后验分布为：

$$p(Z=k/\vec{\boldsymbol{x}}_i)=\frac{p_\alpha(k)p(\vec{\boldsymbol{x}}_i/Z=k)}{p_{\mathcal{M}}(\vec{\boldsymbol{x}}_i)}=\frac{\alpha_k p(\vec{\boldsymbol{x}}_i \mid \vec{\mu}_k,\Sigma_k)}{\sum_{l=1}^K \alpha_l p(\vec{\boldsymbol{x}}_i \mid \vec{\mu}_l,\Sigma_l)}$$

即所有导致输出为 $\vec{\boldsymbol{x}}_i$ 的情况下，$Z=k$ 发生的概率。

若已知高斯混合分布，则高斯混合聚类的原理是：如果样本 $\vec{\boldsymbol{x}}_i$ 最有可能是 $Z=k$ 产生的，则可将该样本划归到簇 C_k。即通过最大后验概率确定样本所属的聚类。

用数学语言表述为：高斯混合聚类将样本集 D 划分成 K 个簇 $\mathcal{C}=\{C_1,C_2,\cdots,C_K\}$ 。对于每个样本 $\vec{\boldsymbol{x}}_i$ ，其簇标记 λ_i 为：

$$\lambda_i=\arg\max_k p(Z=k/\vec{\boldsymbol{x}}_i), k=1,2,\cdots,K$$

现在的问题是，如何学习高斯混合分布的参数。由于涉及隐变量 Z，故可以采用EM算法求解。

6.2.4 密度聚类

密度聚类density-based clustering假设聚类结构能够通过样本分布的紧密程度来确定。DBSCAN是常用的密度聚类算法，它通过一组邻域参数 $(\epsilon, MinPts)$ 来描述样本分布的紧密程度。给定数据集 $D=\{\vec{\boldsymbol{x}}_1,\vec{\boldsymbol{x}}_2,\cdots,\vec{\boldsymbol{x}}_N\}$，数据集属性定义如下。

- ϵ -邻域：$N_\epsilon(\vec{\boldsymbol{x}}_i)=\{\vec{\boldsymbol{x}}_j \in D \mid distance(\vec{\boldsymbol{x}}_i,\vec{\boldsymbol{x}}_j)\leqslant \epsilon\}$，
  $N_\epsilon(\vec{\boldsymbol{x}}_i)$ 包含了样本集 D 中与 $\vec{\boldsymbol{x}}_i$ 距离不大于 ϵ 的所有的样本。
- 核心对象core object：若 $|N_\epsilon(\vec{\boldsymbol{x}}_i)|\geqslant MinPts$，则称 $\vec{\boldsymbol{x}}_i$ 是一个核心对象。
  即：若 $\vec{\boldsymbol{x}}_i$ 的 ϵ -邻域中至少包含 $MinPts$ 个样本，则 $\vec{\boldsymbol{x}}_i$ 是一个核心对象。
- 密度直达directyl density-reachable：若 $\vec{\boldsymbol{x}}_i$ 是一个核心对象，且 $\vec{\boldsymbol{x}}_j \in N_\epsilon(\vec{\boldsymbol{x}}_i)$，则称 $\vec{\boldsymbol{x}}_j$ 由 $\vec{\boldsymbol{x}}_i$ 密度直达，记作 $\vec{\boldsymbol{x}}_i \mapsto \vec{\boldsymbol{x}}_j$。
- 密度可达density-reachable：对于 $\vec{\boldsymbol{x}}_i$ 和 $\vec{\boldsymbol{x}}_j$，若存在样本序列 $(\vec{\boldsymbol{p}}_0,\vec{\boldsymbol{p}}_1,\vec{\boldsymbol{p}}_2,\cdots,\vec{\boldsymbol{p}}_m,\vec{\boldsymbol{p}}_{m+1})$，其中 $\vec{\boldsymbol{p}}_0=\vec{\boldsymbol{x}}_i,\vec{\boldsymbol{p}}_{m+1}=\vec{\boldsymbol{x}}_j,\vec{\boldsymbol{p}}_s \in D, s=1,2,\cdots,m$。如果 $\vec{\boldsymbol{p}}_{s+1}$ 由 $\vec{\boldsymbol{p}}_s, s=0,1,2,\cdots,m$ 密度直达，则称 $\vec{\boldsymbol{x}}_j$ 由 $\vec{\boldsymbol{x}}_i$ 密度可达，记作 $\vec{\boldsymbol{x}}_i \rightsquigarrow \vec{\boldsymbol{x}}_j$。
- 密度相连density-connected：对于 $\vec{\boldsymbol{x}}_i$ 和 $\vec{\boldsymbol{x}}_j$，若存在 $\vec{\boldsymbol{x}}_k$ ，使得 $\vec{\boldsymbol{x}}_i$ 与 $\vec{\boldsymbol{x}}_j$ 均由 $\vec{\boldsymbol{x}}_k$ 密度可达，则称 $\vec{\boldsymbol{x}}_i$ 与 $\vec{\boldsymbol{x}}_j$ 密度相连，记作 $\vec{\boldsymbol{x}}_i \sim \vec{\boldsymbol{x}}_j$。

DBSCAN算法的簇定义：给定邻域参数 $(\epsilon, MinPts)$ ，一个簇 $C \subseteq D$ 是满足下列性质的非空样本子集。

- 连接性 connectivity：若 $\vec{\boldsymbol{x}}_i \in C, \vec{\boldsymbol{x}}_j \in C$，则 $\vec{\boldsymbol{x}}_i \sim \vec{\boldsymbol{x}}_j$。
- 最大性maximality：若 $\vec{\boldsymbol{x}}_i \in C$，且 $\vec{\boldsymbol{x}}_i \rightsquigarrow \vec{\boldsymbol{x}}_j$，则 $\vec{\boldsymbol{x}}_j \in C$。

即一个簇是由密度可达关系导出的最大的密度相连样本集合。

DBSCAN算法的思想：若 $\vec{\boldsymbol{x}}$ 为核心对象，则 $\vec{\boldsymbol{x}}$ 密度可达的所有样本组成的集合记作 $\boldsymbol{X} = \{\vec{\boldsymbol{x}}' \in D \mid \vec{\boldsymbol{x}} \rightsquigarrow \vec{\boldsymbol{x}}'\}$，可以证明 $\boldsymbol{X}$ 就是满足连接性与最大性的簇。于是 DBSCAN算法首先任选数据集中的一个核心对象作为种子seed，再由此出发确定相应的聚类簇。

下面给出DBSCAN算法。

- 输入
  - 数据集 $D = \{\vec{\boldsymbol{x}}_1, \vec{\boldsymbol{x}}_2, \cdots, \vec{\boldsymbol{x}}_N\}$。
  - 邻域参数 $(\epsilon, MinPts)$。
- 输出：簇划分 $\mathcal{C} = \{C_1, C_2, \cdots, C_K\}$。
- 算法步骤如下。
  - 初始化核心对象集合为空集：$\Omega = \phi$。
  - 寻找核心对象：遍历所有的样本点 $\vec{\boldsymbol{x}}_i, i = 1, 2, \cdots, N$，计算 $N_\epsilon(\vec{\boldsymbol{x}}_i)$。如果 $|N_\epsilon(\vec{\boldsymbol{x}}_i)| \geqslant MinPts$，则 $\Omega = \Omega \bigcup \{\vec{\boldsymbol{x}}_i\}$。
  - 迭代：以任一未访问过的核心对象为出发点，找出有其密度可达的样本生成的聚类簇，直到所有核心对象都被访问为止。

6.2.5 层次聚类

层次聚类（hierarchical clustering）可在不同层上对数据集进行划分，形成树状的聚类结构。AGglomerative NESting（AGNES）是一种常用的层次聚类算法。

AGNES算法原理：AGNES首先将数据集中的每个样本看作一个初始的聚类簇，然后再不断地找出距离最近的两个聚类簇进行合并。就这样不断地合并直到达到预设的聚类簇的个数。这里的关键在于：如何计算聚类簇之间的距离？

由于每个簇就是一个集合，因此需要给出集合之间的距离。给定聚类簇 C_i, C_j，有如下三种距离。

- 最小距离：

$$d_{min}(C_i, C_j) = \min_{\vec{\boldsymbol{x}}_i \in C_i, \vec{\boldsymbol{x}}_j \in C_j} distance(\vec{\boldsymbol{x}}_i, \vec{\boldsymbol{x}}_j)$$

  它是两个簇的样本对之间距离的最小值。
- 最大距离：

$$d_{max}(C_i, C_j) = \max_{\vec{\boldsymbol{x}}_i \in C_i, \vec{\boldsymbol{x}}_j \in C_j} distance(\vec{\boldsymbol{x}}_i, \vec{\boldsymbol{x}}_j)$$

  它是两个簇的样本对之间距离的最大值。

- 平均距离：

$$d_{avg}(C_i, C_j) = \frac{1}{|C_i||C_j|} \sum_{\vec{x}_i \in C_i} \sum_{\vec{x}_j \in C_j} distance(\vec{x}_i, \vec{x}_j)$$

  它是两个簇的样本对之间距离的平均值。

当AGNES算法的聚类簇距离采用 d_{min} 时，称为单链接single-linkage算法；当AGNES算法的聚类簇距离采用 d_{max} 时，称为全链接complete-linkage算法；当AGNES算法的聚类簇距离采用 d_{avg} 时，称为均链接average-linkage算法。

下面给出AGNES算法。

- 输入
  - 数据集 $D = \{\vec{x}_1, \vec{x}_2, \cdots, \vec{x}_N\}$。
  - 聚类簇距离度量函数 d。
  - 聚类簇数量 K。
- 输出：簇划分 $\mathcal{C} = \{C_1, C_2, \cdots, C_K\}$。
- 算法步骤如下。
  - 初始化：将每个样本都作为一个簇

$$C_i = \{\vec{x}_i\}, i = 1, 2, \cdots, N$$

  - 迭代，终止条件为聚类簇的数量为 K。迭代过程如下：
    - 计算聚类簇之间的距离，找出距离最近的两个簇，将这两个簇合并。

6.2.6 EM 算法

三硬币模型

EM算法也称为期望极大算法。它是一种迭代算法，用于含有隐变量的概率模型参数估计。EM算法的每次迭代由两步组成：E步求期望；M步求极大。这也是EM这个名词的由来。

通常对于概率模型的参数估计，可以使用极大似然估计法或者贝叶斯估计法来估计模型参数。但是当模型含有隐变量时，这种方法就行不通了，如下面的三硬币模型。

三硬币模型：已知三枚硬币 A，B，C。这些硬币正面出现的概率分别为 π, p, q，现进行如下的试验：

- 先投掷硬币 A，若是正面则选硬币 B；若是反面则选硬币 C 。
- 然后投掷被选出来的硬币，投掷的结果如果是正面则记作 1；投掷的结果如果是反面则记作0。

- ❑ 独立重复地 n 次试验，观测结果为：`1,1,0,1,0,...0,1`。

  现在只能观测到投掷硬币最终的结果，无法观测投掷硬币的中间过程，求估计参数 π, p, q。

设随机变量 y 是观测变量，表示一次试验观察到的结果是 1 或者0；随机变量 z 是隐变量，表示未观测到的投掷硬币 A 的结果；$\theta = (\pi, p, q)$ 是模型参数，则：

$$P(y;\theta) = \sum_{Z} P(y, z;\theta) = \sum_{z} P(z;\theta)P(y/z;\theta) = \pi p^{y}(1-p)^{1-y} + (1-\pi)q^{y}(1-q)^{1-y}$$

> 随机变量 y 的数据可以观测，而随机变量 z 的数据不可观测。

将观测数据表示为 $Y_{seq} = (y_1, y_2, \cdots, y_n)$，未观测数据表示为 $Z_{seq} = (z_1, z_2, \cdots, z_n)$。则由于每次实验都是独立重复地进行的，于是：

$$P(Y_{seq};\theta) = \prod_{j=1}^{n} P(y_j;\theta) = \prod_{j=1}^{n} [\pi p^{y_j}(1-p)^{1-y_j} + (1-\pi)q^{y_j}(1-q)^{1-y_j}]$$

考虑求模型参数 $\theta = (\pi, p, q)$ 的极大似然估计，即：

$$\hat{\theta} = \arg\max_{\theta} \log P(Y_{seq};\theta)$$

这个问题没有解析解，但可以通过EM算法迭代求解。

EM算法先给出参数的初值：$\theta^{<0>} = (\pi^{<0>}, p^{<0>}, q^{<0>})$。设第 i 次迭代参数的估计值为：$\theta^{<i>} = (\pi^{<i>}, p^{<i>}, q^{<i>})$，则第 $i+1$ 次迭代如下。

- ❑ E步：计算模型在参数 $\theta^{<i>} = (\pi^{<i>}, p^{<i>}, q^{<i>})$ 下，观测数据 y_j 来自于投掷硬币 B 的概率：

$$\mu^{<i+1>} = \frac{\pi^{<i>}(p^{<i>})^{y_j}(1-p^{<i>})^{1-y_j}}{\pi^{<i>}(p^{<i>})^{y_j}(1-p^{<i>})^{1-y_j} + (1-\pi^{<i>})(q^{<i>})^{y_j}(1-q^{<i>})^{1-y_j}}$$

> 它其实就是 $P(z=1/y=y_j)$，即：已知观测变量 $y = y_j$ 的条件下，隐变量 $z = 1$ 的后验概率。

- ❑ M 步：更新模型参数的新估计值如下。

$$\pi^{<i+1>} = \frac{1}{n}\sum_{j=1}^{n}\mu_j^{<i+1>}$$

$$p^{<i+1>} = \frac{\sum_{j=1}^{n}\mu_j^{<i+1>}y_j}{\sum_{j=1}^{n}\mu_j^{<i+1>}}$$

$$q^{<i+1>} = \frac{\sum_{j=1}^{n}(1-\mu_j^{<i+1>})y_j}{\sum_{j=1}^{n}(1-\mu_j^{<i+1>})}$$

注意：EM算法与初值的选择有关，选择不同的初始值可能得到不同的参数估计值。

算法原理

令 Y 表示观测随机变量的数据，Z 表示隐随机变量的数据。Y 和 Z 连在一起称为完全数据，观测数据 Y 又称为不完全数据。

- 假设给定观测数据 Y ，其概率分布为 $P(Y;\theta)$，其中 θ 是需要估计的模型参数，则不完全数据 Y 的似然函数是 $P(Y;\theta)$，对数似然函数为 $L(\theta)=\log P(Y;\theta)$。
- 假定 Y 和 Z 的联合概率分布是 $P(Y,Z;\theta)$,则完全数据的对数似然函数是 $\log P(Y,Z;\theta)$。

EM 算法原理：EM 算法通过迭代求解的是 $L(\theta) = \log P(Y;\theta) = \log\sum_Z P(Y,Z;\theta) = \log\left[\sum_Z P(Y/Z;\theta)P(Z;\theta)\right]$ 的极大值。困难在于：该目标函数包含了未观测数据分布的积分和对数。EM 算法的目标是：通过迭代逐步近似极大化 $L(\theta)$。

假设在第 i 次迭代后，θ 的估计是：$\theta^{<i>}$。希望新估计值 θ 能够使得 $L(\theta)$ 增加。考虑：

$$L(\theta)-L(\theta^{<i>}) = \log\left[\sum_Z P(Y/Z;\theta)P(Z;\theta)\right] - \log P(Y;\theta^{<i>})$$

> 这里因为 $\theta^{<i>}$ 已知，所以 $\log P(Y;\theta^{<i>})$ 可以直接计算得出。

- 利用 Jensen不等式：

$$\log\sum_j \lambda_j y_j \geqslant \sum_j \lambda_j \log y_j,\quad \lambda_j \geqslant 0,\quad \sum_j \lambda_j = 1$$

考虑到 $\sum_Z P(Z/Y;\theta^{<i>})=1$，则有：

$$
\begin{aligned}
L(\theta)-L(\theta^{<i>}) &= \log\left[\sum_Z P(Z/Y;\theta^{<i>})\frac{P(Y/Z;\theta)P(Z;\theta)}{P(Z/Y;\theta^{<i>})}\right]-\log P(Y;\theta^{<i>})\\
&\geqslant \sum_Z\left[P(Z/Y;\theta^{<i>})\log\frac{P(Y/Z;\theta)P(Z;\theta)}{P(Z/Y;\theta^{<i>})}\right]-\log P(Y;\theta^{<i>})\\
&= \sum_Z\left[P(Z/Y;\theta^{<i>})\log\frac{P(Y/Z;\theta)P(Z;\theta)}{P(Z/Y;\theta^{<i>})}\right]\\
&\quad -\sum_Z\left[P(Z/Y;\theta^{<i>})\log P(Y;\theta^{<i>})\right]\\
&= \sum_Z\left[P(Z/Y;\theta^{<i>})\log\frac{P(Y/Z;\theta)P(Z;\theta)}{P(Z/Y;\theta^{<i>})P(Y;\theta^{<i>})}\right]
\end{aligned}
$$

- 令：

$$
B(\theta,\theta^{<i>})=L(\theta^{<i>})+\sum_Z\left[P(Z/Y;\theta^{<i>})\log\frac{P(Y/Z;\theta)P(Z;\theta)}{P(Z/Y;\theta^{<i>})P(Y;\theta^{<i>})}\right]
$$

则有：$L(\theta)\geqslant B(\theta,\theta^{<i>})$，因此 $B(\theta,\theta^{<i>})$ 是 $L(\theta^{<i>})$ 的一个下界。

  - 根据定义有：$L(\theta^{<i>})=B(\theta^{<i>},\theta^{<i>})$。

  > 因为此时有：$\frac{P(Y/Z;\theta^{<i>})P(Z;\theta^{<i>})}{P(Z/Y;\theta^{<i>})P(Y;\theta^{<i>})}=\frac{P(Y,Z;\theta^{<i>})}{P(Y,Z;\theta^{<i>})}=1$

  - 任何可以使得 $B(\theta,\theta^{<i>})$ 增大的 θ，也可以使 $L(\theta)$ 增大。为了使得 $L(\theta)$ 尽可能地增大，则选择使得 $B(\theta,\theta^{<i>})$ 取极大值的 θ $\theta^{<i+1>}=\arg\max_\theta B(\theta,\theta^{<i>})$。

- 求极大值：

$$
\begin{aligned}
\theta^{<i+1>} &= \arg\max_\theta B(\theta,\theta^{<i>})\\
&= \arg\max_\theta\left[L(\theta^{<i>})+\sum_Z P(Z/Y;\theta^{<i>})\log\frac{P(Y/Z;\theta)P(Z;\theta)}{P(Z/Y;\theta^{<i>})P(Y;\theta^{<i>})}\right]\\
&= \arg\max_\theta\left[\sum_Z P(Z/Y;\theta^{<i>})\log(P(Y/Z;\theta)P(Z;\theta))\right]\\
&= \arg\max_\theta\left[\sum_Z P(Z/Y;\theta^{<i>})\log P(Y,Z;\theta)\right]
\end{aligned}
$$

> 其中，$L(\theta^{<i>}),\sum_Z P(Z/Y;\theta^{<i>})\log P(Z/Y;\theta^{<i>})P(Y;\theta^{<i>})$ 与 θ 无关，因此省略。
>
> 最后一步是因为：$P(Y,Z;\theta)=P(Y/Z;\theta)P(Z;\theta)$。

EM 算法不能保证得到全局最优值。其优点在于：简单性、普适性。

给出 EM 算法如下。

- 输入
  - 观测变量数据 Y。
  - 隐变量数据 Z。
  - 联合分布 $P(Y, Z; \theta)$。
  - 条件分布 $P(Z/Y; \theta)$。
- 输出：模型参数 θ。
- 算法步骤
  - 选择参数的初值 $\theta^{<0>}$，开始迭代。

    > 参数的初始条件可以任意选择，但是 EM 算法对初值是敏感的。

  - 反复迭代直到收敛，迭代步骤如下。
    设 $\theta^{<i>}$ 为第 i 次迭代参数 θ 的估计值，则在第 $i+1$ 步可以进行以下两种计算。
    - E步。求期望，计算：

$$Q(\theta, \theta^{<i>}) = E_Z[\log P(Y, Z; \theta)/Y; \theta^{<i>}] = \sum_Z \log P(Y, Z; \theta) P(Z/Y; \theta^{<i>})$$

      这里 $P(Z/Y; \theta^{<i>})$ 表示在给定观测数据 Y 和当前的参数估计 $\theta^{<i>}$ 的条件下，隐变量 Z 的条件概率分布。它是真实的 Z 的条件概率分布的一个估计。
      $Q(\theta, \theta^{<i>})$ 是算法的核心，称为 Q 函数。它的物理意义是：完全数据的对数似然函数 $\log P(Y, Z; \theta)$ 关于某个分布的期望。该分布就是 Z 的真实条件概率分布的一个近似，由 $P(Z/Y; \theta^{<i>})$ 给出。
      在 $Q(\theta, \theta^{<i>})$ 中，第一个变元表示要极大化的参数；第二个变元表示参数的当前估计值。
    - M步。更新参数：

$$\theta^{<i+1>} = \arg\max_{\theta} Q(\theta, \theta^{<i>})$$

> 通常收敛的条件是：给定较小的正数 $\varepsilon_1, \varepsilon_2$，满足 $||\theta^{<i+1>} - \theta^{<i>}|| < \varepsilon_1$，或者 $||Q(\theta^{<i+1>}, \theta^{<i>}) - Q(\theta^{<i>}, \theta^{<i>})|| < \varepsilon_2$。

EM算法的直观理解：EM算法的目标是最大化对数似然函数 $L(\theta) = \log P(Y)$。但是直接求解这个目标是有问题的。因为要求解该目标，首先要得到未观测数据的条件分布 $P(Z/Y; \theta)$；但是未观测数据的条件分布就是你待求目标参数 θ 的解的函数。这是一个“鸡生蛋-蛋生鸡”的问题。EM算法试图多次猜测这个未观测数据条件的分布 $P(Z/Y; \theta)$（每一轮迭代都猜测一个参数值 $\theta^{<i>}$，而每一轮参数值都对应着一个未观测数据的条件分布 $P(Z/Y; \theta^{<i>})$，因为未

观测数据的分布就是参数的函数），然后通过最大化某个变量来求解参数值。这个变量就是 $B(\theta,\theta^{<i>})$ 变量，它是真实的似然函数的下界。

- 如果猜测正确，则 B 就是真实的似然函数。
- 如果猜测不正确，则 B 就是真实似然函数的一个下界。

> 为什么用 $P(Z/Y;\theta)$ 而不用 $P(Y/Z;\theta)$?
> 因为 $P(Z/Y;\theta)$ 是在已知 Y 的情况下，Z 的分布。实际情况确实是 Y 已知，但是 Z 的分布未知。
> $P(Y/Z;\theta)$ 是在已知 Z 的情况下，Y 的分布。真实情况是：Y 已经知晓，但是 Z 未知，因此定义无法满足。

EM算法收敛性定理有如下二个。

- 定理一：设 $P(Y;\theta)$ 为观测数据的似然函数，$\theta^{<i>},\quad i=1,2,\cdots$ 为EM算法得到的参数估计序列，$P(Y;\theta^{<i>}),\quad i=1,2,\cdots$ 为对应的似然函数序列，则 $P(Y;\theta^{<i>}),\quad i=1,2,\cdots$ 是单调递增的，即 $P(Y;\theta^{<i+1>})\geqslant P(Y;\theta^{<i>})$。
- 定理二：设 $L(\theta)=\log P(Y;\theta)$ 为观测数据的对数似然函数，$\theta^{<i>},\quad i=1,2,\cdots$ 为EM算法得到的参数估计序列，$L(\theta^{<i>}),\quad i=1,2,\cdots$ 为对应的对数似然函数序列。
  - 如果 $P(Y;\theta)$ 有上界，则 $L(\theta^{<i>}),\quad i=1,2,\cdots$ 收敛到某一个值 L^*。
  - 在函数 $Q(\theta,\theta^{<i>})$ 与 $L(\theta)$ 满足一定条件下，由 EM 算法得到的参数估计序列 $\theta^{<i>},i=1,2,\cdots$ 的收敛值 θ^* 是 $L(\theta)$ 的稳定点。

> 关于定理二中，“满足一定条件”：大多数条件下其实都是满足的。
> 定理二只能保证参数估计序列收敛到对数似然函数序列的稳定点 L^*，不能保证收敛到极大值点。

EM算法的收敛性蕴含了两层意义：对数似然函数序列 $L(\theta^{<i>}),\quad i=1,2,\cdots$ 收敛；参数估计序列 $\theta^{<i>},\quad i=1,2,\cdots$ 收敛。前者并不蕴含后者。

EM算法初值的选择非常重要，常用的办法是给出一批初值，然后分别从每个初值开始使用EM算法。最后对得到的各个估计值加以比较，从中选择对数似然函数最大的那个。

EM 算法求解高斯混合模型

定义高斯混合分布如下：

$$p_{\mathcal{M}}(\vec{\boldsymbol{x}})=\sum_{k=1}^{K}\alpha_k p(\vec{\boldsymbol{x}}\mid\vec{\mu}_k,\Sigma_k)$$

高斯混合分布由 K 个成分混合而成，其中每个成分对应一个高斯分布。$\vec{\mu}_k, \Sigma_k$ 是第 k 个成分对应的高斯分布的参数。$\alpha_k > 0$ 是第 k 个成分的混合系数，且 $\sum_{k=1}^{K} \alpha_k = 1$。

假设样本训练集 $D = \{\vec{x}_1, \vec{x}_2, \cdots, \vec{x}_N\}$ 的生成过程是由高斯混合分布给出的。令：随机变量 $Z \in \{1, 2, \cdots, K\}$，$Z$ 的概率分布 $p(Z = k) = \alpha_k$，记作 $p_\alpha(k)$。生成样本的过程分为两步：

- 首先根据概率分布 $p_\alpha(k)$ 生成随机变量 Z；
- 再根据 Z 的结果，比如 $Z = 2$，根据概率 $p(\vec{x} \mid \vec{\mu}_2, \Sigma_2)$ 生成样本。

假设观测数据 $\vec{x}_1, \vec{x}_2, \cdots, \vec{x}_N$ 是由高斯混合分布生成的。观察数据 $\vec{x}_j, j = 1, 2, \cdots, N$ 是已知的，但是它来自哪个分模型是未知的。对观察数据 $\vec{x}_j$ 定义 K 个隐变量 γ_{jk} 为：

$$\gamma_{jk} = \begin{cases} 1, & \text{if } \vec{x}_j \text{ comes from } p(\vec{x} \mid \vec{\mu}_k, \Sigma_k) \\ 0, & \text{else} \end{cases} \quad j = 1, 2, \cdots, N; \quad k = 1, 2, \cdots, K$$

则 γ_{jk} 是 0-1 随机变量，其为 1 的概率为 α_k。于是完全数据为：$(\vec{x}_j, \gamma_{j1}, \gamma_{j2}, \cdots, \gamma_{jK}), \quad j = 1, 2, \cdots, N$。

需要求解的参数为：$\theta = (\alpha_1, \cdots, \alpha_K, \vec{\mu}_1, \cdots, \vec{\mu}_K, \Sigma_1, \cdots, \Sigma_K)$。完全数据的似然函数为：

$$\begin{aligned}
P(D, \gamma; \theta) &= \prod_{j=1}^{N} P(\vec{x}_j, \gamma_{j1}, \gamma_{j2}, \cdots, \gamma_{jK}; \theta) \\
&= \prod_{k=1}^{K} \prod_{j=1}^{N} [\alpha_k p(\vec{x}_j \mid \vec{\mu}_k, \Sigma_k)]^{\gamma_{jk}} \\
&= \prod_{k=1}^{K} \alpha_k^{n_k} \prod_{j=1}^{N} [p(\vec{x}_j \mid \vec{\mu}_k, \Sigma_k)]^{\gamma_{jk}} \\
&= \prod_{k=1}^{K} \alpha_k^{n_k} \prod_{j=1}^{N} \left[\frac{1}{(2\pi)^{n/2} |\Sigma_k|^{1/2}} \exp\left(-\frac{1}{2} (\vec{x}_j - \vec{\mu}_k)^T \Sigma_k^{-1} (\vec{x}_j - \vec{\mu}_k) \right) \right]^{\gamma_{jk}}
\end{aligned}$$

其中，$n_k = \sum_{j=1}^{N} \gamma_{jk}, \sum_{k=1}^{K} n_k = N$。$n_k$ 表示在所有的观测数据中，产生自第 k 个分模型的观测数据的数量。

> 这里有个基本假设：对不同的 j，γ_{jk} 是相互独立的。

完全数据的对数似然函数为：

$$\log P(D, \gamma; \theta) = \sum_{k=1}^{K} \left\{ n_k \log \alpha_k + \sum_{j=1}^{N} \gamma_{jk} \left[-\frac{n}{2} \log(2\pi) - \frac{1}{2} \log |\Sigma_k| - \frac{1}{2} (\vec{x}_j - \vec{\mu}_k)^T \Sigma_k^{-1} (\vec{x}_j - \vec{\mu}_k) \right] \right\}$$

- EM算法的E步。确定 Q 函数：

$$Q(\theta,\theta^{<i>}) = E_\gamma\left[\log P(\vec{\boldsymbol{x}}_j,\gamma;\theta)/\vec{\boldsymbol{x}}=\vec{\boldsymbol{x}}_j;\theta^{<i>}\right] = \sum_\gamma \log P(\vec{\boldsymbol{x}}_j,\gamma;\theta)P(\gamma/\vec{\boldsymbol{x}}=\vec{\boldsymbol{x}}_j;\theta^{<i>})$$
$$= \sum_{k=1}^{K}\left\{n_k\log\alpha_k + \sum_{j=1}^{N} E\gamma_{jk}\left[-\frac{n}{2}\log(2\pi)-\frac{1}{2}\log|\Sigma_k|-\frac{1}{2}(\vec{\boldsymbol{x}}_j-\vec{\mu}_k)^T\Sigma_k^{-1}(\vec{\boldsymbol{x}}_j-\vec{\mu}_k)\right]\right\}$$

  - 令 $\hat{\gamma}_{jk}=E\gamma_{jk}=E(\gamma_{jk}/\vec{\boldsymbol{x}}=\vec{\boldsymbol{x}}_j;\theta^{<i>})$，则

$$\begin{aligned}\hat{\gamma}_{jk} &= P(\gamma_{jk}=1/\vec{\boldsymbol{x}}=\vec{\boldsymbol{x}}_j;\theta^{<i>})\cdot 1+0\\ &= \frac{P(\gamma_{jk}=1,\vec{\boldsymbol{x}}_j;\theta^{<i>})}{\sum_{k=1}^{K}P(\gamma_{jk}=1,\vec{\boldsymbol{x}}_j;\theta^{<i>})}\\ &= \frac{P(\vec{\boldsymbol{x}}_j/\gamma_{jk}=1;\theta^{<i>})P(\gamma_{jk}=1;\theta^{<i>})}{\sum_{k=1}^{K}P(\vec{\boldsymbol{x}}_j/\gamma_{jk}=1;\theta^{<i>})P(\gamma_{jk}=1;\theta^{<i>})} \quad j=1,2,\cdots,N;\quad k=1,2,\cdots,K\\ &= \frac{\alpha_k^{<i>}p(\vec{\boldsymbol{x}}_j\mid\vec{\mu}_k^{<i>},\Sigma_k^{<i>})}{\sum_{l=1}^{K}\alpha_l^{<i>}p(\vec{\boldsymbol{x}}_j\mid\vec{\mu}_l^{<i>},\Sigma_l^{<i>})}\end{aligned}$$

  $\hat{\gamma}_{jk}$ 是当前模型参数 $\theta^{<i>}$ 下第 j 个观测数据来自第 k 个分模型的概率，称为分模型 k 对观测数据 y_j 的响应度。

  - 于是有：

$$Q(\theta,\theta^{<i>}) = \sum_{k=1}^{K}\left\{n_k\log\alpha_k + \sum_{j=1}^{N}\hat{\gamma}_{jk}\left[-\frac{n}{2}\log(2\pi)-\frac{1}{2}\log|\Sigma_k|-\frac{1}{2}(\vec{\boldsymbol{x}}_j-\vec{\mu}_k)^T\Sigma_k^{-1}(\vec{\boldsymbol{x}}_j-\vec{\mu}_k)\right]\right\}$$

- EM算法的M步。求解 $Q(\theta,\theta^{<i>})$ 对 θ 的极大值：

$$\theta^{<i+1>} = \arg\max_\theta Q(\theta,\theta^{<i>})$$

根据偏导数为 0，以及 $\sum_{k=1}^{K}\alpha_k=1$ 得到

$$\vec{\mu}_k^{<i+1>} = \frac{\sum_{j=1}^{N}\hat{\gamma}_{jk}\vec{\boldsymbol{x}}_j}{\sum_{j=1}^{N}\hat{\gamma}_{jk}},\quad k=1,2,\cdots,K$$
$$\Sigma_k^{<i+1>} = \frac{\sum_{j=1}^{N}\hat{\gamma}_{jk}(\vec{\boldsymbol{x}}_j-\vec{\mu}_k^{<i+1>})(\vec{\boldsymbol{x}}_j-\vec{\mu}_k^{<i+1>})^T}{\sum_{j=1}^{N}\hat{\gamma}_{jk}},\quad k=1,2,\cdots,K$$

$$\alpha_k^{<i+1>} = \frac{n_k}{N} = \frac{\sum_{j=1}^{N} \hat{\gamma}_{jk}}{N}, \quad k = 1, 2, \cdots, K$$

- 重复上述计算，直到对数似然函数值不再有明显的变化为止。

下面给出高斯混合模型参数估计的EM算法。

- 输入
  - 观察数据 $D = \{\vec{\boldsymbol{x}}_1, \vec{\boldsymbol{x}}_2, \cdots, \vec{\boldsymbol{x}}_N\}$。
  - 高斯混合成分个数 K。
- 输出：高斯混合模型参数。
- 算法步骤
  - 取参数的初始值 $\theta^{<0>} = (\alpha_1^{<0>}, \cdots, \alpha_K^{<0>}, \vec{\mu}_1^{<0>}, \cdots, \vec{\mu}_K^{<0>}, \Sigma_1^{<0>}, \cdots, \Sigma_K^{<0>})$。
  - 迭代直至算法收敛。迭代过程如下。
    - E步：根据当前模型参数，计算分模型 k 对观测数据 $\vec{\boldsymbol{x}}_j$ 的响应度：

$$\hat{\gamma}_{jk} = \frac{\alpha_k^{<i>} p(\vec{\boldsymbol{x}}_j \mid \vec{\mu}_k^{<i>}, \Sigma_k^{<i>})}{\sum_{l=1}^{K} \alpha_l^{<i>} p(\vec{\boldsymbol{x}}_j \mid \vec{\mu}_l^{<i>}, \Sigma_l^{<i>})} \qquad j = 1, 2, \cdots, N; \quad k = 1, 2, \cdots, K$$

    - M步。计算新一轮迭代的模型参数：

$$\vec{\mu}_k^{<i+1>} = \frac{\sum_{j=1}^{N} \hat{\gamma}_{jk} \vec{\boldsymbol{x}}_j}{\sum_{j=1}^{N} \hat{\gamma}_{jk}}, \quad k = 1, 2, \cdots, K$$

$$\Sigma_k^{<i+1>} = \frac{\sum_{j=1}^{N} \hat{\gamma}_{jk} (\vec{\boldsymbol{x}}_j - \vec{\mu}_k^{<i+1>})(\vec{\boldsymbol{x}}_j - \vec{\mu}_k^{<i+1>})^T}{\sum_{j=1}^{N} \hat{\gamma}_{jk}}, \quad k = 1, 2, \cdots, K$$

$$\alpha_k^{<i+1>} = \frac{n_k}{N} = \frac{\sum_{j=1}^{N} \hat{\gamma}_{jk}}{N}, \quad k = 1, 2, \cdots, K$$

6.2.7 实际中的聚类要求

上面介绍了多种聚类算法，其实不同的聚类算法有不同的应用背景，有的适合于大数据集，有的适合小数据集；有的适合处理批量数据，有的适合处理增量数据；有的可以善于发现任意形状的聚簇，有的算法思想简单适用于小数据集。总的来说，机器学习中针对聚类的典型要求包括如下 8 个。

（1）可伸缩性：当数据量从几百上升到几千万时，聚类结果的准确度能一致。

（2）不同类型属性的处理能力：即许多算法针对的数值类型的数据。但是，实际应用场景中，尤其是大数据应用场合，经常会遇到二元类型数据，分类/标称类型数据，序数型数据。

（3）发现任意形状的类簇：许多聚类算法基于距离（欧式距离或曼哈顿距离）来量化对象之间的相似度。基于这种方式，我们往往只能发现相似尺寸和密度的球状类簇或者凸

型类簇。但是，实际中类簇的形状可能是任意的，因此，需要根据实际数据类组簇的形状选择合适的算法。

（4）初始化参数：很多算法需要用户提供一定个数的初始参数，比如期望的类簇个数，类簇初始中心点的设定。聚类的结果对这些参数十分敏感，调参数需要大量的人力负担，也非常影响聚类结果的准确性。实际中，很多聚类问题，一些参数是事先不知道的，如类簇个数。

（5）算法的抗噪能力：噪声数据通常可以理解为影响聚类结果的干扰数据，包含孤立点，错误数据等，一些算法对这些噪声数据非常敏感，抗噪性能差，会导致低质量的聚类。

（6）增量聚类和对输入次序的敏感度：一些算法不能将新加入的数据快速插入到已有的聚类结果中，还有一些算法针对不同次序的数据输入，产生的聚类结果差异很大。

（7）高维处理能力：有些算法只能处理低维度数据，而处理高维数据的能力很弱，高维空间中的数据分布十分稀疏，且高度倾斜。

（8）结果的可解释性和可用性：我们希望得到的聚类结果都能用特定的语义、知识进行解释，并和实际的应用场景相联系。

6.3 Python 实战

首先导入包：

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets.samples_generator import make_blobs
from sklearn import  cluster
from sklearn.metrics import adjusted_rand_score
from sklearn import mixture
```

然后给出产生数据的函数：

```
def create_data(centers,num=100,std=0.7):
    X, labels_true = make_blobs(n_samples=num, centers=centers, cluster_std=std)
    return  X,labels_true
```

- ❑ 参数
  - ❍ centers：聚类的中心点组成的数组。如果中心点是二维的，则产生的每个样本都是二维的。
  - ❍ num：样本数。
  - ❍ std：每个簇中样本的标准差。
- ❑ 返回值：一个元组，第一个元素为样本点，第二个元素为样本点的真实簇分类标记。

该函数主要用到make_blobs函数，该函数产生的是分隔的高斯分布的聚类簇。

下面观察一下生成的样本点。先给出函数：

```
def plot_data(*data):
    X,labels_true=data
    labels=np.unique(labels_true)
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    colors='rgbyckm'
    for i,label in enumerate(labels):
        position=labels_true==label
        ax.scatter(X[position,0],X[position,1],label="cluster %d"%label,
        color=colors[i%len(colors)])

    ax.legend(loc="best",framealpha=0.5)
    ax.set_xlabel("X[0]")
    ax.set_ylabel("Y[1]")
    ax.set_title("data")
    plt.show()
```

然后调用该函数观察 1000 个点的分布：

```
X,labels_true=create_data([[1,1],[2,2],[1,2],[10,20]],1000,0.5)
plot_data(X,labels_true)
```

结果如图 6.1 所示。随机产生了四个簇。为了考察聚类性能，将三个簇交织在一起，另一个簇则较远。

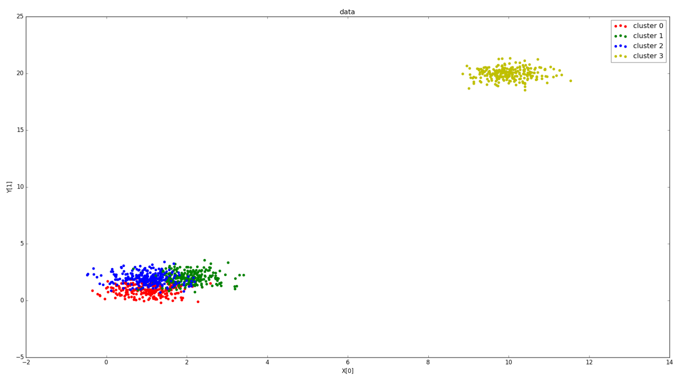


图 6.1 cluster_data

6.3.1 K 均值聚类（KMeans）

KMeans是 scikit-learn提供的 k 均值聚类算法模型，其原型为：

```
class sklearn.cluster.KMeans(n_clusters=8, init='k-means++', n_init=10, max_iter=300,
 tol=0.0001, precompute_distances='auto', verbose=0, random_state=None, copy_x=True,
n_jobs=1)
```

参数

- ❑ n_clusters：一个整数，指定分类簇的数量。
- ❑ init：一个字符串，指定初始均值向量的策略。可以为如下。
  - ❍ 'k-means++'：该初始化策略选择的初始均值向量相互之间都距离较远，它的效果较好。
  - ❍ 'random'：从数据集中随机选择 K 个样本作为初始均值向量。
  - ❍ 或者提供一个数组，数组的形状为 (n_clusters.n_features)，该数组作为初始均值向量。

> K均值算法总能够收敛，但是其收敛情况高度依赖于初始化的均值。有可能收敛到局部极小值。因此通常都是用多组初始均值向量来计算若干次，选择其中最优的那一次。而k-means++策略选择的初始均值向量可以在一定程度上解决这个问题。

- ❑ n_init：一个整数，指定了k均值算法运行的次数。每一次都会选择一组不同的初始化均值向量，最终算法会选择最佳的分类簇来作为最终的结果。
- ❑ max_iter：一个整数，指定了单轮k均值算法中，最大的迭代次数。算法总的最大迭代次数为 max_iter * n_init。
- ❑ precompute_distances：可以为布尔值或者字符串 'auto'。该参数指定是否提前计算好样本之间的距离（如果提前计算距离，则需要更多的内存，但是算法会运行得更快）。
  - ❍ 'auto'：如果 n_samples*n_clusters > 12 million，则不提前计算；
  - ❍ True：总是提前计算；
  - ❍ False：总是不提前计算。
- ❑ tol：一个浮点数，指定了算法收敛的阈值。
- ❑ n_jobs：一个正数。指定任务并形时指定的 CPU 数量。如果为 −1 则使用所有可用的 CPU。
- ❑ verbose：一个整数。如果为 0，则不输出日志信息；如果为 1，则每隔一段时间打印一次日志信息；如果大于 1，则打印日志信息更频繁。
- ❑ random_state：一个整数或者一个RandomState实例，或者None。
  - ❍ 如果为整数，则它指定了随机数生成器的种子。
  - ❍ 如果为RandomState实例，则指定了随机数生成器。
  - ❍ 如果为None，则使用默认的随机数生成器。
- ❑ copy_x：布尔值，主要用于precompute_distances=True的情况。
  - ❍ 如果为True，则预计算距离的时候，并不修改原始数据。
  - ❍ 如果为False，则预计算距离的时候，会修改原始数据用于节省内存；然后当算法结束的时候，会将原始数据返还。但是可能会因为浮点数的表示，会有一些精度误差。

属性

- cluster_centers_：给出分类簇的均值向量。
- labels_：给出了每个样本所属的簇的标记。
- inertia_：给出了每个样本距离它们各自最近的簇中心的距离之和。

方法

- fit(X[,y])：训练模型。
- fit_predict(X[, y])：训练模型并预测每个样本所属的簇。它等价于先调用fit方法，后调用predict方法。
- predict(X)：预测样本所属的簇。
- score(X[, y])：给出了样本距离个簇中心的偏移量的相反数。

首先使用KMeans来考察聚类结果，给出测试函数：

```
def test_Kmeans(*data):
    X,labels_true=data
    clst=cluster.KMeans()
    clst.fit(X)
    predicted_labels=clst.predict(X)
    print("ARI:%s"% adjusted_rand_score(labels_true,predicted_labels))
    print("Sum center distance %s"%clst.inertia_)
```

然后调用test_Kmeans：

```
centers=[[1,1],[2,2],[1,2],[10,20]]
X,labels_true=create_data(centers,1000,0.5)
test_Kmeans(X,labels_true)
```

结果如下：

```
ARI:0.37248223219179594
Sum center distance 232.143136059
```

其中ARI指标为0.37248223219179594（越大越好），所有样本距离各簇中心点的距离之和为232.143136059。

考察簇的数量的影响。给出测试函数：

```
def test_Kmeans_nclusters(*data):
    X,labels_true=data
    nums=range(1,50)
    ARIs=[]
    Distances=[]
    for num in nums:
        clst=cluster.KMeans(n_clusters=num)
        clst.fit(X)
```

```
        predicted_labels=clst.predict(X)
        ARIs.append(adjusted_rand_score(labels_true,predicted_labels))
        Distances.append(clst.inertia_)

    ## 绘图
    fig=plt.figure()
    ax=fig.add_subplot(1,2,1)
    ax.plot(nums,ARIs,marker="+")
    ax.set_xlabel("n_clusters")
    ax.set_ylabel("ARI")
    ax=fig.add_subplot(1,2,2)
    ax.plot(nums,Distances,marker='o')
    ax.set_xlabel("n_clusters")
    ax.set_ylabel("inertia_")
    fig.suptitle("KMeans")
    plt.show()
```

然后同样调用test_Kmeans_nclusters，结果如图 6.2 所示。可以看到 n_clusters=4时，ARI指数最大（因为确实是从四个中心点产生四个簇）。而inertia_（所有样本距离各簇中心点的距离之和）在 n_clusters=1时最大。因为产生的测试数据有三个中心点较近，一个中心点较远。因此如果指定KMeans算法的簇的数量为 1 （即所有样本都划归为一个簇），则确实样本离簇中心的距离之和最大。

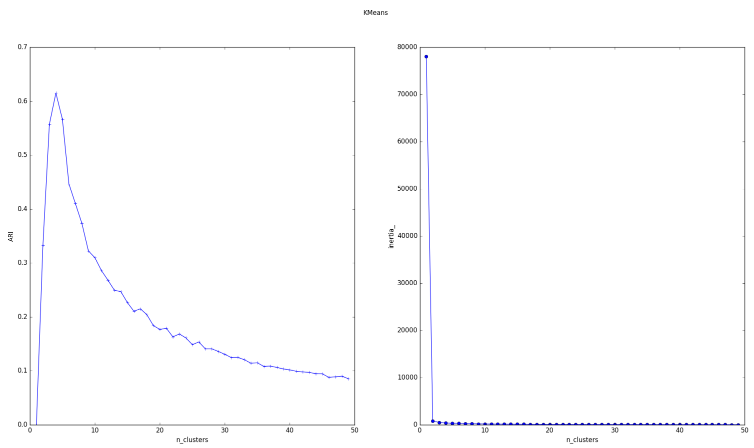


图 6.2 Kmeans_nclusters

考察k均值算法运行的次数和选择初始中心向量策略的影响，给出测试函数：

```
def test_Kmeans_n_init(*data):
    X,labels_true=data
    nums=range(1,50)
    ## 绘图
    fig=plt.figure()
```

```
ARIs_k=[]
Distances_k=[]
ARIs_r=[]
Distances_r=[]
for num in nums:
        clst=cluster.KMeans(n_init=num,init='k-means++')
        clst.fit(X)
        predicted_labels=clst.predict(X)
        ARIs_k.append(adjusted_rand_score(labels_true,predicted_labels))
        Distances_k.append(clst.inertia_)

        clst=cluster.KMeans(n_init=num,init='random')
        clst.fit(X)
        predicted_labels=clst.predict(X)
        ARIs_r.append(adjusted_rand_score(labels_true,predicted_labels))
        Distances_r.append(clst.inertia_)

ax=fig.add_subplot(1,2,1)
ax.plot(nums,ARIs_k,marker="+",label="k-means++")
ax.plot(nums,ARIs_r,marker="+",label="random")
ax.set_xlabel("n_init")
ax.set_ylabel("ARI")
ax.set_ylim(0,1)
ax.legend(loc='best')
ax=fig.add_subplot(1,2,2)
ax.plot(nums,Distances_k,marker='o',label="k-means++")
ax.plot(nums,Distances_r,marker='o',label="random")
ax.set_xlabel("n_init")
ax.set_ylabel("inertia_")
ax.legend(loc='best')

fig.suptitle("KMeans")
plt.show()
```

然后同样调用test_Kmeans_n_init，结果如图 6.3 所示。可以看到ARI指数与inertia_总距离随 n_init震荡。因此这两项指标与n_init影响不是很大。而且随机选择和使用 'k-means++'策略选择初始中心向量，对于聚类效果的影响也不大。

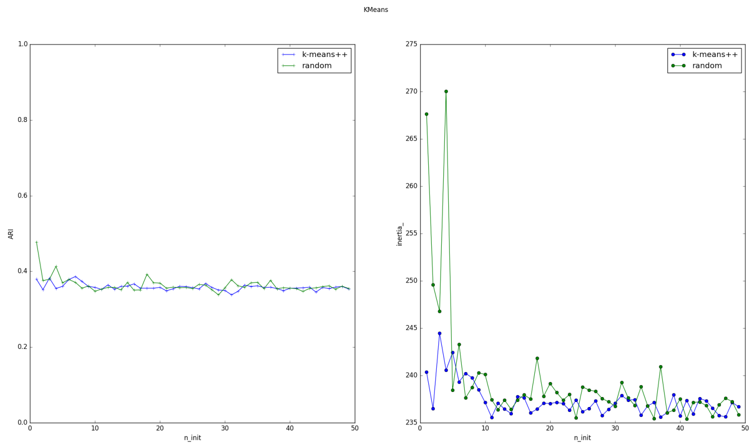


图 6.3 Kmeans_n_init

6.3.2 密度聚类（DBSCAN）

DBSCAN是scikit-learn提供的密度聚类算法模型。其原型为：

```
class sklearn.cluster.DBSCAN(eps=0.5, min_samples=5, metric='euclidean',
algorithm='auto', leaf_size=30, p=None, random_state=None)
```

参数

- eps：ϵ 参数，用于确定邻域大小。
- min_samples：$MinPts$ 参数，用于判断核心对象。
- metric：一个字符串或者可调用对象，用于计算距离。如果是字符串，则必须是在 metrics.pairwise.calculate_distance 中指定。
- algorithm：一个字符串，用于计算两点间距离并找出最近邻的点，可以为如下。
  - 'auto'：由算法自动选取合适的算法。
  - 'ball_tree'：用ball树来搜索。
  - 'kd_tree'：用kd树来搜索。
  - 'brute'：暴力搜索。
- leaf_size：一个整数，用于指定当algorithm=ball_tree或者kd_tree时，树的叶节点大小。该参数会影响构建树、搜索最近邻的速度，同时影响存储树的内存。
- random_state：被废弃的接口，将在scikit-learn v 0.18中移除。

属性

- core_sample_indices_：核心样本在原始训练集中的位置。
- components_：核心样本的一份副本。
- labels_：每个样本所属的簇标记。对于噪声样本，其簇标记为 -1 副本。

方法

- ❑ fit(X[, y, sample_weight])：训练模型。
- ❑ fit_predict(X[, y, sample_weight])：训练模型并预测每个样本所属的簇标记。

首先使用DBSCAN来考察聚类结果，给出测试函数：

```
def test_DBSCAN(*data):
    X,labels_true=data
    clst=cluster.DBSCAN()
    predicted_labels=clst.fit_predict(X)
    print("ARI:%s"% adjusted_rand_score(labels_true,predicted_labels))
    print("Core sample num:%d"%len(clst.core_sample_indices_))
```

然后调用test_DBSCAN：

```
centers=[[1,1],[2,2],[1,2],[10,20]]
X,labels_true=create_data(centers,1000,0.5)
test_DBSCAN(X,labels_true)
```

结果如下：

```
ARI:0.330288936333354167
Core sample num:989
```

其中ARI指标为0.330288936333354167（越大越好）。DBSCAN根据密度，将原始数据集划分为 989 个簇。

然后考察 ϵ 参数的影响：

```
def test_DBSCAN_epsilon(*data):
    X,labels_true=data
    epsilons=np.logspace(-1,1.5)
    ARIs=[]
    Core_nums=[]
    for epsilon in epsilons:
        clst=cluster.DBSCAN(eps=epsilon)
        predicted_labels=clst.fit_predict(X)
        ARIs.append( adjusted_rand_score(labels_true,predicted_labels))
        Core_nums.append(len(clst.core_sample_indices_))

    ## 绘图
    fig=plt.figure()
    ax=fig.add_subplot(1,2,1)
    ax.plot(epsilons,ARIs,marker='+')
    ax.set_xscale('log')
    ax.set_xlabel(r"$\epsilon$")
    ax.set_ylim(0,1)
```

```
    ax.set_ylabel('ARI')

    ax=fig.add_subplot(1,2,2)
    ax.plot(epsilons,Core_nums,marker='o')
    ax.set_xscale('log')
    ax.set_xlabel(r"$\epsilon$")
    ax.set_ylabel('Core_Nums')

    fig.suptitle("DBSCAN")
    plt.show()
```

然后同样调用test_DBSCAN_epsilon，结果如图 6.4 所示。可以看到ARI指数随着 ϵ 的增长，先上升后保持平稳，最后断崖式下降。断崖式下降是因为我们产生的训练样本的间距比较小，最远的两个样本点之间的距离不超过 30，当 ϵ 过大时，所有的点都在一个邻域中。

核心样本数量随着 ϵ 的增长而上升，这是因为随着 ϵ 的增长，样本点的邻域在扩展，则样本点邻域内的样本会更多，这就产生了更多满足条件的核心样本点。但是样本集中的样本数量有限，因此核心样本点的数量增长到一定数目后会趋于稳定。

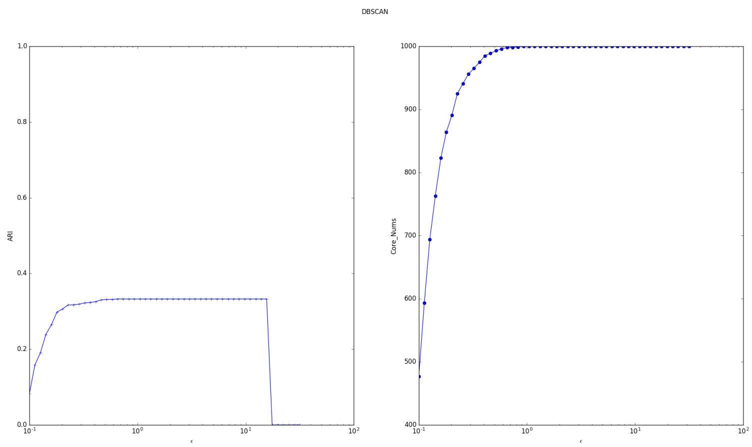


图 6.4 DBSCAN_epsilon

然后考察 $MinPts$ 参数的影响：

```
def test_DBSCAN_min_samples(*data):
    X,labels_true=data
    min_samples=range(1,100)
    ARIs=[]
    Core_nums=[]
    for num in min_samples:
        clst=cluster.DBSCAN(min_samples=num)
        predicted_labels=clst.fit_predict(X)
        ARIs.append( adjusted_rand_score(labels_true,predicted_labels))
        Core_nums.append(len(clst.core_sample_indices_))
```

```
## 绘图
fig=plt.figure()
ax=fig.add_subplot(1,2,1)
ax.plot(min_samples,ARIs,marker='+')
ax.set_xlabel( "min_samples")
ax.set_ylim(0,1)
ax.set_ylabel('ARI')

ax=fig.add_subplot(1,2,2)
ax.plot(min_samples,Core_nums,marker='o')
ax.set_xlabel( "min_samples")
ax.set_ylabel('Core_Nums')

fig.suptitle("DBSCAN")
plt.show()
```

再然后同样调用test_DBSCAN_min_samples，结果如图 6.5 所示。可以看到ARI指数随着 $MinPts$ 的增长，平稳地下降。而核心样本数量随着 $MinPts$ 的增长基本上呈线性下降，这是因为随着 $MinPts$ 的增长，样本点的邻域中必须包含更多的样本才能使它成为一个核心样本点。因此产生的核心样本点越来越少。

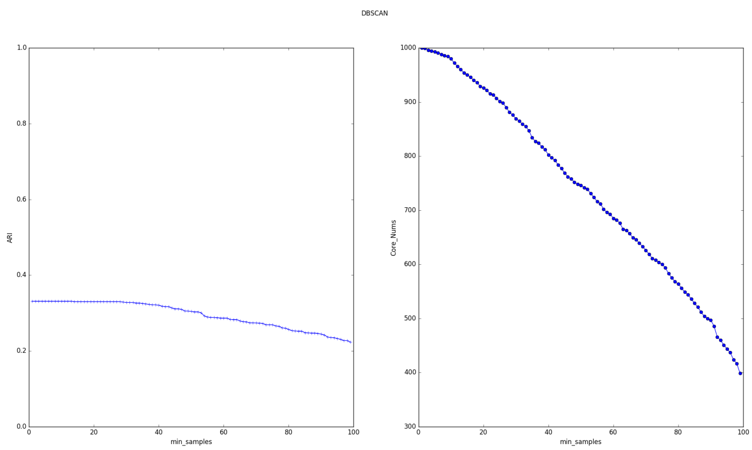


图 6.5 DBSCAN_min_samples

6.3.3 层次聚类（AgglomerativeClustering）

AgglomerativeClustering是scikit-learn提供的层次聚类算法模型。其原型为：

```
class sklearn.cluster.AgglomerativeClustering(n_clusters=2, affinity='euclidean',
 memory=Memory(cachedir=None), connectivity=None, n_components=None,
 compute_full_tree='auto', linkage='ward', pooling_func=<function mean>)
```

参数

- n_clusters：一个整数，指定分类簇的数量。
- connectivity：一个数组或者可调用对象或者为None，用于指定连接矩阵。它给出了每个样本的可连接样本。
- affinity：一个字符串或者可调用对象，用于计算距离。可以为：'euclidean', 'l1', 'l2', 'manhattan', 'cosine', 'precomputed'，如果linkage='ward'，则'affinity必须是'euclidean'。
- memory：用于缓存输出的结果，默认为不缓存。
- n_components：将在scikit-learn v 0.18中移除。
- compute_full_tree：通常当训练了n_clusters之后，训练过程就停止。但是如果 compute_full_tree=True，则会继续训练从而生成一颗完整的树。
- linkage：一个字符串，用于指定链接算法。
  - 'ward'：单链接single-linkage算法，采用 d_{min}。
  - 'complete'：全链接complete-linkage算法，采用 d_{max}。
  - 'average'：均链接average-linkage算法, 采用 d_{avg}。
- pooling_func：一个可调用对象，它的输入是一组特征的值，输出是一个数值。

属性

- labels_：每个样本的簇标记。
- n_leaves_：分层树的叶结点数量。
- n_components_：连接图中连通分量的估计值。
- children_：一个数组，给出了每个非叶结点中的子节点数量。

方法

- fit(X[, y])：训练模型。
- fit_predict(X[, y])：训练模型并预测每个样本所属的簇标记。

首先使用AgglomerativeClustering来考察聚类结果，给出测试函数：

```
def test_AgglomerativeClustering(*data):
    X,labels_true=data
    clst=cluster.AgglomerativeClustering()
    predicted_labels=clst.fit_predict(X)
    print("ARI:%s"% adjusted_rand_score(labels_true,predicted_labels))
```

然后调用test_AgglomerativeClustering：

```
centers=[[1,1],[2,2],[1,2],[10,20]]
X,labels_true=create_data(centers,1000,0.5)
test_AgglomerativeClustering(X,labels_true)
```

结果如下，其中ARI指标为0.33266533066132264（越大越好）。

```
ARI:0.33266533066132264
```

接下来考察簇的数量对于聚类效果的影响，首先给出测试函数：

```
def test_AgglomerativeClustering_nclusters(*data):
    X,labels_true=data
    nums=range(1,50)
    ARIs=[]
    for num in nums:
        clst=cluster.AgglomerativeClustering(n_clusters=num)
        predicted_labels=clst.fit_predict(X)
        ARIs.append(adjusted_rand_score(labels_true,predicted_labels))

    ## 绘图
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    ax.plot(nums,ARIs,marker="+")
    ax.set_xlabel("n_clusters")
    ax.set_ylabel("ARI")
    fig.suptitle("AgglomerativeClustering")
    plt.show()
```

然后同样调用test_AgglomerativeClustering_nclusters，结果如图 6.6 所示。可以看到 n_clusters=4 时，ARI指数最大（因为确实是从四个中心点产生的四个簇）。

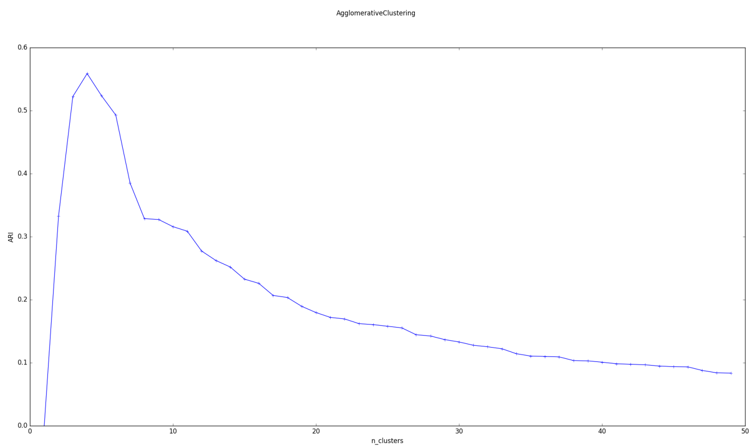


图 6.6　AgglomerativeClustering_nclusters

再然后考察链接方式的影响，给出测试函数如下：

```
def test_AgglomerativeClustering_linkage(*data):
    X,labels_true=data
    nums=range(1,50)
    fig=plt.figure()
```

```
    ax=fig.add_subplot(1,1,1)

    linkages=['ward','complete','average']
    markers="+o*"
    for i, linkage in enumerate(linkages):
        ARIs=[]
        for num in nums:
            clst=cluster.AgglomerativeClustering(n_clusters=num,linkage=linkage)
            predicted_labels=clst.fit_predict(X)
            ARIs.append(adjusted_rand_score(labels_true,predicted_labels))
        ax.plot(nums,ARIs,marker=markers[i],label="linkage:%s"%linkage)

    ax.set_xlabel("n_clusters")
    ax.set_ylabel("ARI")
    ax.legend(loc="best")
    fig.suptitle("AgglomerativeClustering")
    plt.show()
```

然后同样调用test_AgglomerativeClustering_linkage，结果如图 6.7 所示。可以看到，三种链接方式随分类簇的数量的总体趋势都相差无几。但是单链接方式ward的峰值最大，且峰值最大的分类簇的数量刚好等于实际上生成样本的簇的数量。

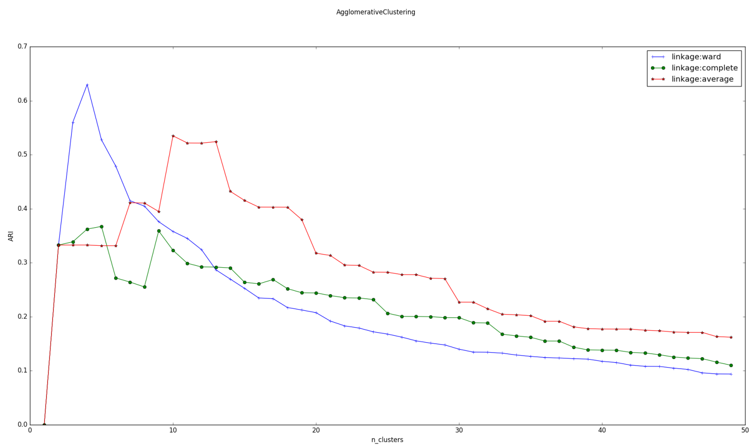


图 6.7 AgglomerativeClustering_linkage

6.3.4 混合高斯（GaussianMixture）模型

GMM 是scikit-learn给出的混合高斯模型，其原型为：

```
class sklearn.mixture.GaussianMixture(n_components=1, covariance_type='full',
 tol=0.001, reg_covar=1e-06, max_iter=100, n_init=1, init_params='kmeans',
 weights_init=None, means_init=None, precisions_init=None, random_state=None,
 warm_start=False, verbose=0, verbose_interval=10)
```

参数

- ❑ n_components：一个整数，指定分模型的数量，默认为 1。
- ❑ covariance_type：一个字符串，指定协方差的类型。必须为下列值之一。
  - ❍ 'spherical' ：球状型，每个分模型的协方差矩阵都是一个标量值。
  - ❍ 'tied'：结点型，所有的分模型都共享一个协方差矩阵。
  - ❍ 'diag'：对角型，每个分模型的协方差矩阵都是对角矩阵。
  - ❍ 'full'：全型，每个分模型都有自己的协方差矩阵。
- ❑ random_state：一个整数或者一个RandomState实例，或者None。
  - ❍ 如果为整数，则它指定了随机数生成器的种子。
  - ❍ 如果为RandomState实例，则指定了随机数生成器。
  - ❍ 如果为None，则使用默认的随机数生成器。
- ❑ reg_covar：一个浮点数，添加到协方差矩阵对角线上元素，确保所有的协方差都是正数。
- ❑ tol：一个浮点数，用于指定收敛的阈值。EM迭代算法中，当对数似然函数平均增益低于此阈值时，迭代停止。
- ❑ max_iter：一个整数，指定EM算法迭代的次数。
- ❑ n_init：一个整数，用于指定初始化次数。即算法会运行多轮，只有表现最好的哪个结果作为最终的结果。
- ❑ weights-init：一个序列，形状为(n_components,)，指定初始化的权重。
- ❑ means_init：一个数组，形状为(n_components,n_features)，指定初始化的均值。
- ❑ precision_init：用户提供的初始precisions（协方差矩阵的逆矩阵），根据 covariance_type 的不同，该参数值的形状不同。
- ❑ init_params：一个字符串，可以为'kmeans'/'random'，用于指定初始化权重的策略。
- ❑ verbose：一个整数。如果为 0，则不输出日志信息；如果为 1，则每隔一段时间打印一次日志信息；如果大于 1，则打印日志信息更频繁。
- ❑ warm_start：一个布尔值。如果为True，则上一次训练的结果将作为本次训练的开始条件。
- ❑ verbose_interval：一个整数，指定输出日志的间隔。

属性

- ❑ weights_：一个数组，形状为(n_components,)。该属性存储了每个分模型的权重。
- ❑ means_：一个数组，形状为(n_components, n_features)。该属性存储了每个分模型的均值 μ_k。

方法

- ❑ fit(X[, y])：训练模型。
- ❑ fit_predict(X[, y])：训练模型并预测每个样本所属的簇标记。

- ❑ predict(X)：预测样本所属的簇标记。
- ❑ predict_proba(X)：预测样本所属各个簇的概率。
- ❑ sample([n_samples, random_state])：根据模型来随机生成一组样本。
- ❑ score(X[, y])：计算模型在样本总体上的对数似然函数。
- ❑ score_samples(X)：给出每个样本的对数似然函数。

首先使用GMM来考察聚类结果，给出测试函数：

```
def test_GMM(*data):
    X,labels_true=data
    clst=mixture.GaussianMixture()
    predicted_labels=clst.fit_predict(X)
    print("ARI:%s"% adjusted_rand_score(labels_true,predicted_labels))
```

然后调用test_GMM：

```
centers=[[1,1],[2,2],[1,2],[10,20]]
X,labels_true=create_data(centers,1000,0.5)
test_GMM(X,labels_true)
```

结果如下：

```
ARI:0.0
```

其中ARI指标为0（越大越好）。这是因为默认的GMM只有一个簇，无论哪个样本点，都是划归到该簇。

然后考察簇的数量n_components对于聚类效果的影响，给出测试函数：

```
def test_GMM_n_components(*data):
    X,labels_true=data
    nums=range(1,50)
    ARIs=[]
    for num in nums:
        clst=mixture.GaussianMixture(n_components=num)
        predicted_labels=clst.fit_predict(X)
        ARIs.append(adjusted_rand_score(labels_true,predicted_labels))

    ## 绘图
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    ax.plot(nums,ARIs,marker="+")
    ax.set_xlabel("n_components")
    ax.set_ylabel("ARI")
    fig.suptitle("GMM")
    plt.show()
```

同样调用test_GMM_n_components，结果如图 6.8 所示。可以看到，对于高斯混合模型，可以看到 n_components=4时，ARI指数最大（因为确实是从四个中心点产生的四个簇）。

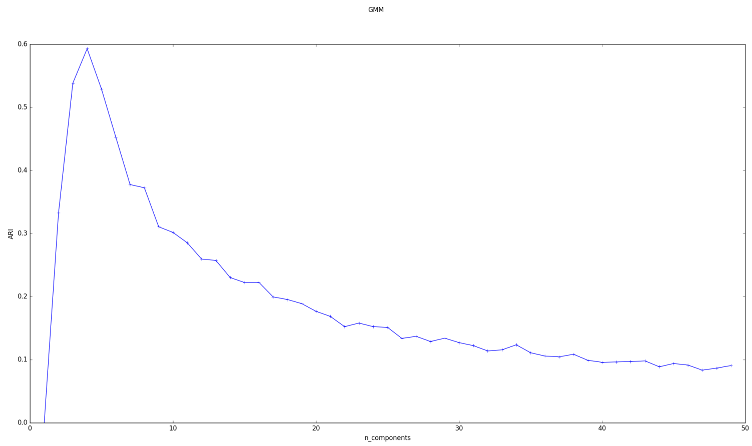


图 6.8 GMM_n_components

最后考察协方差类型的影响，给出测试函数：

```
def test_GMM_cov_type(*data):
    X,labels_true=data
    nums=range(1,50)

    cov_types=['spherical','tied','diag','full']
    markers="+o*s"
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)

    for i ,cov_type in enumerate(cov_types):
        ARIs=[]
        for num in nums:
            clst=mixture.GaussianMixture(n_components=num,covariance_type=cov_type)
            predicted_labels=clst.fit_predict(X)
            ARIs.append(adjusted_rand_score(labels_true,predicted_labels))
        ax.plot(nums,ARIs,marker=markers[i],label="covariance_type:%s"%cov_type)

    ax.set_xlabel("n_components")
    ax.legend(loc="best")
    ax.set_ylabel("ARI")
    fig.suptitle("GMM")
    plt.show()
```

同样调用test_GMM_cov_type，结果如图 6.9 所示。可以看到协方差矩阵的类型对于聚类的效果影响不大。

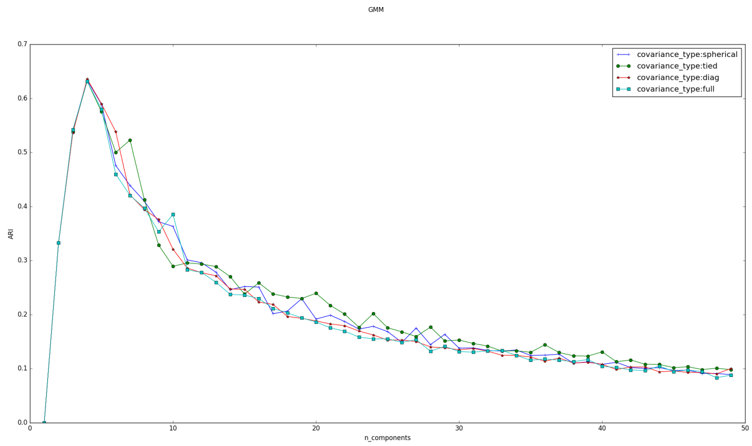


图 6.9 GMM_cov_type

6.4 小结

scikit-learn官方给出了一份各种聚类模型的参数调整和使用场景的建议表格。这里给出如下：

模型	关键参数	使用场景
K 均值算法	簇的数量	通用聚类方法，用于均匀的簇的大小，簇的数量不多的情况
DBSCAN 算法	$\epsilon, MinPts$	用于不均匀的簇大小，以及非平坦的集合结构
AgglomerativeClustering 算法	簇的数量，链接类型	用于簇的数量较多，有连接约束等情况
GMM 算法	一些	用于平坦的集合结构，对密度估计很合适

聚类也许是机器学习算法中“新算法”出现最多、最快的领域。一个重要的原因是聚类不存在客观标准。以我们的例子为例，可以说原始数据集是 4 个簇，因为确实是以 4 个中心点生成了 4 个高斯分布的数据。但是也可以说它们是 2 个簇，因为很明显，这些数据聚集成两团，一个中心点在(10,20)，另一个中心点在(1.5,2)。

在实际应用中，聚类簇的数量的选取通常结合性能度量指标和具体问题分析。如给出了 ARI 随 n_clusters 的曲线。我们可以选择曲线上ARI最大值附近的一批 n_clusters 。然后具体问题具体分析：如果要求每个簇内足够纯净，则倾向于选择较大的 n_clusters，即较大的簇数量（极端情况下，每个样本点就是一个簇，则可以保证每个簇都是纯净的）。如果要求尽可能地将相似的样本划归到一个簇中，则倾向于选择较小的 n_clusters，即较小的簇数量（极端情况下，我们认为所有的样本点都是相似的，则都划归到一个簇中）。

最后，由于实际用可能潜在的簇的数量数目较大，如以百万计，则 ARI 随 n_clusters 的曲线，如果从 0 开始绘制则不现实，因为计算量太大。可以大概估算出簇的数量，如通过降维技术观察原始数据集经过降维后在平面或者三维空间中的分布规律，从中大概估算出簇的量级。或者可以通过他人求解的类似的聚类问题来获取这一类问题近似的簇的量级。

第二篇

机器学习高级篇

第 7 章

支持向量机

7.1 概述

支持向量机（Support Vector Machine，SVM）的基本模型是定义在特征空间上间隔最大的线性分类器。它是一种二类分类模型，当采用了核技巧之后，支持向量机可以用于非线性分类。不同类型的支持向量机解决不同的问题。

- 线性可分支持向量机（也称为硬间隔支持向量机）：当训练数据线性可分时，通过硬间隔最大化，学得一个线性可分支持向量机。
- 线性支持向量机（也称为软间隔支持向量机）：当训练数据近似线性可分时，通过软间隔最大化，学得一个线性支持向量机。
- 非线性支持向量机：当训练数据不可分时，通过使用核技巧以及软间隔最大化，学得一个非线性支持向量机。

在本章中，假设输入空间和特征空间是不同的。通常假设输入空间为欧氏空间，特征空间为希尔伯特空间。此时给定某个输入 $\vec{x}$，通过某种映射（可能为线性映射，也可能为非线性映射）到特征空间的表示为 $\vec{z}$。此时在特征空间中学习线性支持向量机（而不是在输入空间中学习线性支持向量机）。欧氏空间与希尔伯特空间的不同如下：

- 欧氏空间是有限维度的，希尔伯特空间是无穷维度的；
- 欧氏空间 $\subseteq$ 希尔伯特空间 $\subseteq$ 内积空间 $\subseteq$ 赋范空间。

越抽象的空间具有的性质越少，在这样的空间中能得到的结论就越少。不过反过来，如果发现了赋范空间中的某些性质，那么前面那些空间也都具有这个性质。我们生活在三维空间，把它拓展到 n 维空间就是欧氏空间，这是我们比较熟悉的空间，具有一切美好的性质。当我们不局限于有限维度，就来到了希尔伯特空间。从有限到无限是一个质变，很多美好的性质便消失了，一些非常有悖常识的现象会出现。如果再进一步去掉完备性，就来到了内积空间。如果再进一步去掉“角度”的概念，就来到了赋范空间。在这里，起码我们还有“长度”和“距离”的概念。

7.2 算法笔记精华

7.2.1 线性可分支持向量机

原始问题

给定一个特征空间上的训练数据集 $T=\{(\vec{\boldsymbol{x}}_1,y_1),(\vec{\boldsymbol{x}}_2,y_2),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\}$，其中 $\vec{\boldsymbol{x}}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(n)})^T\in\mathcal{X}=\mathbb{R}^n, y_i\in\mathcal{Y}=\{+1,-1\}, i=1,2,\cdots,N$。$\vec{\boldsymbol{x}}_i$ 为第 i 个实例；y_i 为 $\vec{\boldsymbol{x}}_i$ 的类标记：

- 若 $y_i=+1$ 时，则称 $\vec{\boldsymbol{x}}_i$ 为正例；
- 若 $y_i=-1$ 时，则称 $\vec{\boldsymbol{x}}_i$ 为负例。

假设训练数据集线性可分，希望在特征空间中找到一个分离超平面，它能将所有的正例划分到超平面的一侧、所有的负例划分到超平面的另一侧。由于超平面可以用方程 $\vec{\boldsymbol{w}}\cdot\vec{\boldsymbol{x}}+b=0$ 表示，它由法向量 $\vec{\boldsymbol{w}}$ 和截距 b 决定，可以用 $(\vec{\boldsymbol{w}},b)$ 来表示，于是需要求得分离超平面的参数 $(\vec{\boldsymbol{w}},b)$。

当训练数据集线性可分时，理论上存在无穷多个分离超平面可以将两类数据正确分开。线性可分支持向量机提出了间隔最大化这样的约束，最终求得的分离超平面只有唯一的一个。

给定线性可分训练数据集 T，假设通过间隔最大化学习得到的分离超平面为：$\vec{\boldsymbol{w}}^*\cdot\vec{\boldsymbol{x}}+b^*=0$。定义分类决策函数：$f(\vec{\boldsymbol{x}})=\text{sign}(\vec{\boldsymbol{w}}^*\cdot\vec{\boldsymbol{x}}+b^*)$。该分类决策函数也称为线性可分支持向量机。

对于线性可分支持向量机，通常可以将一个样本距离分离超平面的远近来表示分类预测的可靠程度：一个样本距离分离超平面越远，则该样本的分类越可靠；一个样本距离分离超平面越近，则该样本的分类就不那么确信。

给定超平面 $\vec{\boldsymbol{w}}\cdot\vec{\boldsymbol{x}}+b=0$，样本 $\vec{\boldsymbol{x}}_i$ 距超平面的距离为：$|\vec{\boldsymbol{w}}\cdot\vec{\boldsymbol{x}}_i+b|$。$\vec{\boldsymbol{w}}\cdot\vec{\boldsymbol{x}}_i+b$ 的符号与样本标记 y_i 的符号是否一致表示分类是否正确。

- $\vec{w} \cdot \vec{x}_i + b > 0$ 时，即 $\vec{x}_i$ 位于超平面上方，将 $\vec{x}_i$ 划分为正类。若 $y_i = +1$，则分类正确。
- $\vec{w} \cdot \vec{x}_i + b < 0$ 时，即 $\vec{x}_i$ 位于超平面下方，将 $\vec{x}_i$ 划分为负类。若 $y_i = -1$，则分类正确。

所以可以用 $y(\vec{w} \cdot \vec{x} + b)$ 来表示分类的正确性以及确信度（符号决定了正确性，范数决定了确信度）。

给定训练数据集 T，给定超平面 $(\vec{w}, b)$，定义超平面 $(\vec{w}, b)$ 关于样本点 $(\vec{x}_i, y_i)$ 的函数间隔为：$\hat{\gamma}_i = y_i(\vec{w} \cdot \vec{x}_i + b)$。定义超平面 $(\vec{w}, b)$ 关于训练集 T 的函数间隔为超平面 $(\vec{w}, b)$ 关于 T 中所有样本点 $(\vec{x}_i, y_i)$ 的函数间隔之最小值：$\hat{\gamma} = \min_{\vec{x}_i \in T} \hat{\gamma}_i$。

> $\hat{\gamma}_i$ 是关于某个样本点的间隔，$\hat{\gamma}$ 是关于训练集的间隔。

函数间隔存在一个重要的缺陷：当成比例地改变 $\vec{w}$ 和 b，比如将它们改变为 $100\vec{w}$ 和 $100b$，超平面 $100\vec{w} \cdot \vec{x} + 100b = 0$ 不变，但是函数间隔却变为原来的 100 倍。因此我们引入几何间隔。

对于给定的训练数据集 T 和超平面 $(\vec{w}, b)$，定义超平面 $(\vec{w}, b)$ 关于样本点 $(\vec{x}_i, y_i)$ 的几何间隔为：$\gamma_i = y_i(\frac{\vec{w}}{||\vec{w}||} \cdot \vec{x}_i + \frac{b}{||\vec{w}||})$。定义超平面 $(\vec{w}, b)$ 关于训练集 T 的几何间隔为超平面 $(\vec{w}, b)$ 关于 T 中所有样本点 $(\vec{x}_i, y_i)$ 的几何间隔之最小值：$\gamma = \min_{\vec{x}_i \in T} \gamma_i$。

> γ_i 是关于样本点的间隔，γ 是关于训练集的间隔。

支持向量机的目标是：求解能够正确划分训练数据集，且几何间隔最大的分离超平面。这里的几何间隔最大化又称为硬间隔最大化。

这一目标可以用数学语言描述为约束的最优化问题：

$$
\begin{aligned}
&\max_{\vec{w}, b} \gamma \\
&s.t. \quad y_i(\frac{\vec{w}}{||\vec{w}||} \cdot \vec{x}_i + \frac{b}{||\vec{w}||}) \geqslant \gamma, i = 1, 2, \cdots, N
\end{aligned}
$$

根据几何间隔和函数间隔的关系，问题转化为：

$$
\begin{aligned}
&\max_{\vec{w}, b} \frac{\hat{\gamma}}{||\vec{w}||} \\
&s.t. \quad y_i(\vec{w} \cdot \vec{x}_i + b) \geqslant \hat{\gamma}, i = 1, 2, \cdots, N
\end{aligned}
$$

函数间隔 $\hat{\gamma}$ 并不影响最优化问题的解（假设将 $\vec{w}, b$ 按比例地改变为 $k\vec{w}, kb$，此时函数间隔变成 $k\hat{\gamma}$。这一变化对求解最优化问题的不等式约束和最优化目标函数都没有影响），因此令 $\hat{\gamma} = 1$。同时注意到 $\max \frac{1}{||\vec{w}||}$ 等价于 $\min \frac{1}{2}||\vec{w}||_2^2$。于是最优化问题改写为：

$$\min_{\vec{w},b} \frac{1}{2}||\vec{w}||_2^2$$
$$s.t. \quad y_i(\vec{w} \cdot \vec{x}_i + b) - 1 \geqslant 0, i = 1, 2, \cdots, N$$

这是一个凸二次规划问题。

下面给出线性可分支持向量机学习算法——最大间隔法的算法。

- 输入：线性可分训练数据集 $T = \{(\vec{x}_1, y_1), (\vec{x}_2, y_2), \cdots, (\vec{x}_N, y_N)\}$。
- 输出：最大几何间隔的分离超平面和分类决策函数。
- 算法步骤如下。
  - 构造并且求解约束最优化问题：

$$\min_{\vec{w},b} \frac{1}{2}||\vec{w}||_2^2$$
$$s.t. \quad y_i(\vec{w} \cdot \vec{x}_i + b) - 1 \geqslant 0, i = 1, 2, \cdots, N$$

    求得最优解 $\vec{w}^*, b^*$。
  - 由此得到分离超平面：$\vec{w}^* \cdot \vec{x} + b^* = 0$，以及分类决策函数 $f(\vec{x}) = \text{sign}(\vec{w}^* \cdot \vec{x} + b^*)$。

可以证明若训练数据集 T 线性可分，最大间隔分离超平面存在且唯一。

假设已经求得最大间隔分离超平面为 $S = \vec{w}^* \cdot \vec{x} + b^*$。把训练数据集中与 S 距离最近的样本称为支持向量。支持向量就是那些使得约束条件等号成立的样本：即 $y_i(\vec{w}^* \cdot \vec{x}_i + b^*) - 1 = 0$。

- 支持向量中的正例位于超平面 $H_1 : \vec{w}^* \cdot \vec{x} + b^* = 1$。
- 支持向量中的负例位于超平面 $H_2 : \vec{w}^* \cdot \vec{x} + b^* = -1$。

超平面 H_1、H_2 称为间隔边界。H_1、H_2 和最大间隔分离超平面 S 平行，且没有任何实例点落在 H_1、H_2 之间。在 H_1、H_2 之间形成一条隔离带，隔离带的宽度为 $\frac{2}{||\vec{w}^*||}$。

在决定分离超平面时，只有支持向量起作用：

- 如果改变了支持向量，则最大间隔分离超平面也随之改变。
- 如果去掉了间隔边界 H_1 、H_2 之外的任何数量的样本，则最大间隔分离超平面是不变的。

所以支持向量在确定最大间隔分离超平面中起着决定性作用（如图 7.1 所示）。这也是支持向量机名称的由来。

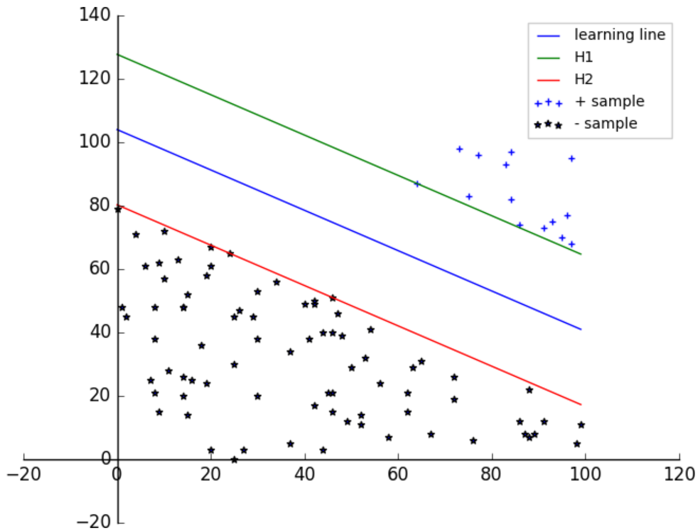


图 7.1 linear_svm

对偶问题

我们将求解线性可分支持向量机的最优化问题作为原始最优化问题。通过应用拉格朗日对偶性转化为对偶问题，这就是线性可分支持向量机的对偶算法。在这一过程中可以方便地引入核函数，从而将支持向量机算法推广到非线性分类问题。

原始问题：

$$\min_{\vec{w},b} \frac{1}{2}||\vec{w}||_2^2$$
$$s.t. \quad y_i(\vec{w}\cdot\vec{x}_i+b)-1\geqslant 0, i=1,2,\cdots,N$$

定义拉格朗日函数：

$$L(\vec{w},b,\vec{\alpha})=\frac{1}{2}||\vec{w}||_2^2-\sum_{i=1}^{N}\alpha_i y_i(\vec{w}\cdot\vec{x}_i+b)+\sum_{i=1}^{N}\alpha_i$$

其中 $\vec{\alpha}=(\alpha_1,\alpha_2,\cdots,\alpha_N)^T$ 为拉格朗日乘子向量。

原始问题的对偶问题是极大极小问题：$\max_{\vec{\alpha}}\min_{\vec{w},b}L(\vec{w},b,\vec{\alpha})$。

先求 $\min_{\vec{w},b}L(\vec{w},b,\vec{\alpha})$。通过拉格朗日函数的偏导数为零，则有：

$$\nabla_{\vec{w}}L(\vec{w},b,\vec{\alpha})=\vec{w}-\sum_{i=1}^{N}\alpha_i y_i\vec{x}_i=\vec{0}$$
$$\nabla_b L(\vec{w},b,\vec{\alpha})=\sum_{i=1}^{N}\alpha_i y_i=0$$

求得 $\vec{\boldsymbol{w}}=\sum_{i=1}^{N}\alpha_i y_i \vec{\boldsymbol{x}}_i$， $\sum_{i=1}^{N}\alpha_i y_i=0$。

再求极大值。将上面得到的 $\vec{\boldsymbol{w}}=\sum_{i=1}^{N}\alpha_i y_i \vec{\boldsymbol{x}}_i$， $\sum_{i=1}^{N}\alpha_i y_i=0$ 代入拉格朗日函数：

$$
\begin{aligned}
L(\vec{\boldsymbol{w}},b,\vec{\alpha})&=\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j(\vec{\boldsymbol{x}}_i\cdot\vec{\boldsymbol{x}}_j)-\sum_{i=1}^{N}\alpha_i y_i\left[(\sum_{j=1}^{N}\alpha_j y_j\vec{\boldsymbol{x}}_j)\cdot\vec{\boldsymbol{x}}_i+b\right]+\sum_{i=1}^{N}\alpha_i\\
&=-\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j(\vec{\boldsymbol{x}}_i\cdot\vec{\boldsymbol{x}}_j)+\sum_{i=1}^{N}\alpha_i
\end{aligned}
$$

于是待求的问题为：

$$
\max_{\vec{\alpha}}-\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j(\vec{\boldsymbol{x}}_i\cdot\vec{\boldsymbol{x}}_j)+\sum_{i=1}^{N}\alpha_i
$$

$$
s.t.\quad \sum_{i=1}^{N}\alpha_i y_i=0
$$

$$
\alpha_i\geqslant 0, i=1,2,\cdots,N
$$

假定已经求得对偶最优化问题的 $\vec{\alpha}$ 的解为 $\vec{\alpha}^*=(\alpha_1^*,\alpha_2^*,\cdots,\alpha_N^*)^T$，则

$$
\vec{\boldsymbol{w}}^*=\sum_{i=1}^{N}\alpha_i^* y_i\vec{\boldsymbol{x}}_i
$$

由于 $\vec{\alpha}^*$ 不是零向量（若它为零向量，则 $\vec{\boldsymbol{w}}^*$ 也为零向量，没有实际应用价值）。则存在某个 j 使得 $\alpha_j^*>0$。根据 $\alpha_j^*(y_j(\vec{\boldsymbol{w}}^*\cdot\vec{\boldsymbol{x}}_j+b^*)-1)=0$（拉格朗日函数极小值条件），此时必有 $y_j(\vec{\boldsymbol{w}}^*\cdot\vec{\boldsymbol{x}}_j+b^*)-1=0$。同时考虑 $y_j^2=1$, 得到：

$$
b^*=y_j-\sum_{i=1}^{N}\alpha_i^* y_i(\vec{\boldsymbol{x}}_i\cdot\vec{\boldsymbol{x}}_j)
$$

于是最大间隔分离超平面为：

$$
\sum_{i=1}^{N}\alpha_i^* y_i(\vec{\boldsymbol{x}}\cdot\vec{\boldsymbol{x}}_i)+b^*=0
$$

分类决策函数为：

$$
f(\vec{\boldsymbol{x}})=\operatorname{sign}(\sum_{i=1}^{N}\alpha_i^* y_i(\vec{\boldsymbol{x}}\cdot\vec{\boldsymbol{x}}_i)+b^*)
$$

上式称为线性可分支持向量机的对偶形式。可以看到 $\vec{\boldsymbol{w}}^*, b^*$ 只依赖于 $\alpha_i^* > 0$ 对应的样本点 $\vec{\boldsymbol{x}}_i, y_i$，因此我们将训练数据集里面对应于 $\alpha_i^* > 0$ 的样本点对应的实例 $\vec{\boldsymbol{x}}_i$ 称为支持向量。

> 对于 $\alpha_i^* > 0$ 的样本点，根据 $\alpha_i^*(y_i(\vec{\boldsymbol{w}}^* \cdot \vec{\boldsymbol{x}}_i + b^*) - 1) = 0$（拉格朗日函数极小值条件），有：$\vec{\boldsymbol{w}}^* \cdot \vec{\boldsymbol{x}}_i + b^* = \pm 1$，即 $\vec{\boldsymbol{x}}_i$ 一定在间隔边界上。这与原始问题给出的支持向量的定义是一致的。

下面给出线性可分支持向量机学习算法的对偶算法。

- 输入：线性可分训练数据集 $T = \{(\vec{\boldsymbol{x}}_1, y_1), (\vec{\boldsymbol{x}}_2, y_2), \cdots, (\vec{\boldsymbol{x}}_N, y_N)\}$。
- 输出：最大几何间隔的分离超平面和分类决策函数。
- 算法步骤如下。
  - 构造并且求解约束最优化问题：

$$\min_{\vec{\alpha}} \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j (\vec{\boldsymbol{x}}_i \cdot \vec{\boldsymbol{x}}_j) - \sum_{i=1}^{N} \alpha_i$$

$$s.t. \quad \sum_{i=1}^{N} \alpha_i y_i = 0$$

$$\alpha_i \geqslant 0, i = 1, 2, \cdots, N$$

    求得最优解 $\vec{\alpha}^* = (\alpha_1^*, \alpha_2^*, \cdots, \alpha_N^*)^T$。
  - 计算

$$\vec{\boldsymbol{w}}^* = \sum_{i=1}^{N} \alpha_i^* y_i \vec{\boldsymbol{x}}_i$$

    同时选择 $\vec{\alpha}^*$ 的一个正的分量 $\alpha_j^* > 0$，计算

$$b^* = y_j - \sum_{i=1}^{N} \alpha_i^* y_i (\vec{\boldsymbol{x}}_i \cdot \vec{\boldsymbol{x}}_j)$$

  - 由此得到最大几何间隔分离超平面：$\vec{\boldsymbol{w}}^* \cdot \vec{\boldsymbol{x}} + b^* = 0$，以及分类决策函数 $f(\vec{\boldsymbol{x}}) = \mathrm{sign}(\vec{\boldsymbol{w}}^* \cdot \vec{\boldsymbol{x}} + b^*)$。

7.2.2　线性支持向量机

对于线性不可分训练数据，线性支持向量机不再适用。但是可以想办法将它扩展到线性不可分问题。

设训练集为 $T=\{(\vec{\boldsymbol{x}}_1,y_1),(\vec{\boldsymbol{x}}_2,y_2),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\}$，其中 $\vec{\boldsymbol{x}}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(n)})^T\in\mathcal{X}=\mathbb{R}^n, y_i\in\mathcal{Y}=\{+1,-1\}, i=1,2,\cdots,N$，假设训练数据集不是线性可分的。这意味着某些样本点 $(\vec{\boldsymbol{x}}_i,y_i)$ 不满足函数间隔大于等于 1 的约束条件。对每个样本点 $(\vec{\boldsymbol{x}}_i,y_i)$ 引进一个松弛变量 $\xi_i\geqslant 0$，修改最优化目标和约束如下。

- 约束条件修改为：$y_i(\vec{\boldsymbol{w}}\cdot\vec{\boldsymbol{x}}_i)\geqslant 1-\xi_i$。表示函数间隔加上松弛变量大于等于 1。
- 优化目标修改为：$\min_{\vec{\boldsymbol{w}},b}\frac{1}{2}||\vec{\boldsymbol{w}}||_2^2+C\sum_{i=1}^N\xi_i$。表示对每个松弛变量 ξ_i ，支付一个代价 $C\xi_i$，这里 $C>0$ 称为惩罚参数。
  - C 值大时，对误分类的惩罚增大，此时误分类点显得更重要。
  - C 值小时，对误分类的惩罚减小，此时误分类点显得不那么重要。

对应于硬间隔最大化，这里称为软间隔最大化。于是线性不可分的线性支持向量机的学习问题就是求解凸二次规划问题：

$$\min_{\vec{\boldsymbol{w}},b,\xi}\left(\frac{1}{2}||\vec{\boldsymbol{w}}||_2^2+C\sum_{i=1}^N\xi_i\right)$$

$$s.t.\quad y_i(\vec{\boldsymbol{w}}\cdot\vec{\boldsymbol{x}}_i+b)\geqslant 1-\xi_i,\quad i=1,2,\cdots,N$$

$$\xi_i\geqslant 0,\quad i=1,2,\cdots,N$$

这称为线性支持向量机的原始问题。可以证明 $\vec{\boldsymbol{w}}$ 的解是唯一的，b 的解不是唯一的，b 的解存在于一个区间内。

假设求解软间隔最大化问题得到的分离超平面为：$\vec{\boldsymbol{w}}^*\cdot\vec{\boldsymbol{x}}+b^*=0$，相应的分类决策函数为 $f(\vec{\boldsymbol{x}})=\text{sign}(\vec{\boldsymbol{w}}^*\cdot\vec{\boldsymbol{x}}+b^*)$。$f(\vec{\boldsymbol{x}})$ 称为线性支持向量机。

对于线性支持向量机的对偶问题，定义拉格朗日函数为：

$$L(\vec{\boldsymbol{w}},b,\vec{\xi},\vec{\alpha},\vec{\mu})=\frac{1}{2}||\vec{\boldsymbol{w}}||_2^2+C\sum_{i=1}^N\xi_i-\sum_{i]1}^N\alpha_i[y_i(\vec{\boldsymbol{w}}_i\cdot\vec{\boldsymbol{x}}_i+b)-1+\xi_i]-\sum_{i=1}^N\mu_i\xi_i$$

$$\alpha_i\geqslant 0,\mu_i\geqslant 0$$

原始问题是拉格朗日函数的极小极大问题；对偶问题是拉格朗日函数的极大极小问题。先求 $L(\vec{\boldsymbol{w}},b,\vec{\xi},\vec{\alpha},\vec{\mu})$ 对 $\vec{\boldsymbol{w}},b,\vec{\xi}$ 的极小。根据偏导数为 0：

$$\nabla_{\vec{\boldsymbol{w}}}L(\vec{\boldsymbol{w}},b,\vec{\xi},\vec{\alpha},\vec{\mu})=\vec{\boldsymbol{w}}-\sum_{i=1}^N\alpha_iy_i\vec{\boldsymbol{x}}_i=\vec{\mathbf{0}}$$

$$\nabla_bL(\vec{\boldsymbol{w}},b,\xi,\alpha,\mu)=-\sum_{i=1}^N\alpha_iy_i=0$$

$$\nabla_{\xi_i}L(\vec{\boldsymbol{w}},b,\xi,\alpha,\mu)=C-\alpha_i-\mu_i=0$$

得到：

$$\vec{\boldsymbol{w}}=\sum_{i=1}^{N}\alpha_i y_i\vec{\boldsymbol{x}}_i$$

$$\sum_{i=1}^{N}\alpha_i y_i=0$$

$$C-\alpha_i-\mu_i=0$$

再求极大问题。将上面三个等式代入拉格朗日函数：

$$\max_{\vec{\alpha},\mu}\min_{\vec{\boldsymbol{w}},b,\vec{\xi}}L(\vec{\boldsymbol{w}},b,\vec{\xi},\vec{\alpha},\vec{\mu})=\max_{\vec{\alpha},\vec{\mu}}\left[-\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j(\vec{\boldsymbol{x}}_i\cdot\vec{\boldsymbol{x}}_j)+\sum_{i=1}^{N}\alpha_i\right]$$

于是得到对偶问题：

$$\min_{\vec{\alpha}}\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j(\vec{\boldsymbol{x}}_i\cdot\vec{\boldsymbol{x}}_j)-\sum_{i=1}^{N}\alpha_i$$

$$s.t.\quad \sum_{i=1}^{N}\alpha_i y_i=0$$

$$0\leqslant\alpha_i\leqslant C,i=1,2,\cdots,N$$

设 $\vec{\alpha}^*=(\alpha_1^*,\alpha_2^*,\cdots,\alpha_N^*)^T$ 是对偶问题的一个解。若存在 $\vec{\alpha}^*$ 的某个分量 $\alpha_j^*,0<\alpha_j^*<C$，则线性支持向量机的原始问题的解可以按照下式得到：

$$\vec{\boldsymbol{w}}^*=\sum_{i=1}^{N}\alpha^* y_i\vec{\boldsymbol{x}}_i$$

$$b^*=y_j-\sum_{i=1}^{N}\alpha^* y_i(\vec{\boldsymbol{x}}_i\cdot\vec{\boldsymbol{x}}_j)$$

于是分离超平面为：$\sum_{i=1}^{N}\alpha^* y_i(\vec{\boldsymbol{x}}_i\cdot\vec{\boldsymbol{x}})+b^*=0$。分类决策函数为：$f(\vec{\boldsymbol{x}})=\text{sign}[\sum_{i=1}^{N}\alpha^* y_i(\vec{\boldsymbol{x}}_i\cdot\vec{\boldsymbol{x}})+b^*]$

下面给出线性支持向量机学习算法的对偶算法。

- 输入
  - 训练数据集 $T=\{(\vec{\boldsymbol{x}}_1,y_1),(\vec{\boldsymbol{x}}_2,y_2),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\}$。
  - 惩罚参数 $C>0$。
- 输出：软间隔最大化分离超平面和分类决策函数。
- 算法步骤如下。

❍ 求解约束最优化问题：

$$\min_{\alpha} \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j(\vec{\boldsymbol{x}}_i \cdot \vec{\boldsymbol{x}}_j) - \sum_{i=1}^{N}\alpha_i$$

$$s.t. \quad \sum_{i=1}^{N}\alpha_i y_i = 0$$

$$C \geqslant \alpha_i \geqslant 0, i = 1, 2, \cdots, N$$

求得最优解 $\vec{\alpha}^* = (\alpha_1^*, \alpha_2^*, \cdots, \alpha_N^*)^T$。

❍ 计算

$$\vec{\boldsymbol{w}}^* = \sum_{i=1}^{N}\alpha_i^* y_i \vec{\boldsymbol{x}}_i$$

同时选择 α^* 的某个分量 $C > \alpha_j^* > 0$，计算

$$b^* = y_j - \sum_{i=1}^{N}\alpha_i^* y_i(\vec{\boldsymbol{x}}_i \cdot \vec{\boldsymbol{x}}_j)$$

> 可能存在多个符合条件的 α_j^*。这是由于原始问题中，对 b 的解不唯一。实际计算时，可以取在所有符合条件的样本点上的平均值。

❍ 由此得到软间隔最大化分离超平面：$\vec{\boldsymbol{w}}^* \cdot \vec{\boldsymbol{x}} + b^* = 0$，以及分类决策函数 $f(\vec{\boldsymbol{x}}) = \text{sign}(\vec{\boldsymbol{w}}^* \cdot \vec{\boldsymbol{x}} + b^*)$。

对偶问题的解 $\vec{\alpha}^* = (\alpha_1^*, \alpha_2^*, \cdots, \alpha_N^*)^T$ 中对应于 $\alpha_i^* > 0$ 的样本点 $(\vec{\boldsymbol{x}}_i, y_i)$ 的实例点 $\vec{\boldsymbol{x}}_i$ 称为支持向量（软间隔的支持向量），可能存在下列情形。

❑ 若 $\alpha_i^* < C$，则松弛量 $\xi_i = 0$，支持向量恰好落在了间隔边界上。
  因为根据 $\nabla_{\xi_i} L(\vec{\boldsymbol{w}}, b, \vec{\xi}, \vec{\alpha}, \vec{\mu}) = C - \alpha_i - \mu_i = 0$，得到：

  ❍ 当 $\alpha_i^* < C$，则 $\mu_i > 0$，根据拉格朗日函数极值条件，必须有 $\xi_i = 0$；
  ❍ 当 $\alpha_i^* = C$，则 $\mu_i = 0$，于是 ξ_i 可能为任何正数。

❑ 若 $\alpha_i^* = C$，且 $0 < \xi_i < 1$，则支持向量落在间隔边界与分离超平面之间，分类正确。
❑ 若 $\alpha_i^* = C$，且 $\xi_i = 1$，则支持向量落在分离超平面上。
❑ 若 $\alpha_i^* = C$，且 $\xi_i > 1$，则支持向量落在分离超平面误分类一侧。

> 根据 ξ_i 的定义，它就是使得函数间隔加上 ξ_i 大于等于 1。即线性可分时，支持向量位于间隔边界：$y_i(\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}}_i + b) = 1$ 上；现在支持向量位于 $y_i(\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}}_i + b) = 1 - \xi_i$ 上。

7.2.3 非线性支持向量机

对于给定的训练集 $T=\{(\vec{\boldsymbol{x}}_1,y_1),(\vec{\boldsymbol{x}}_2,y_2),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\}$，其中 $\vec{\boldsymbol{x}}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(n)})^T\in\mathcal{X}=\mathbb{R}^n,y_i\in\mathcal{Y}=\{+1,-1\},i=1,2,\cdots,N$，如果能用 $\mathbb{R}^n$ 中的一个超曲面将正负实例正确分开，则称这个问题为非线性可分问题。

设 $\mathcal{X}\subseteq\mathbb{R}^n$ 是输入空间，$\mathcal{H}$ 为特征空间（希尔伯特空间）。若存在一个从 $\mathcal{X}$ 到 $\mathcal{H}$ 的映射：$\phi(\vec{\boldsymbol{x}}):\mathcal{X}\rightarrow\mathcal{H}$，使得所有的 $\vec{\boldsymbol{x}},\vec{\boldsymbol{z}}\in\mathcal{X}$，函数 $K(\vec{\boldsymbol{x}},\vec{\boldsymbol{z}})=\phi(\vec{\boldsymbol{x}})\cdot\phi(\vec{\boldsymbol{z}})$，则称 $K(\vec{\boldsymbol{x}},\vec{\boldsymbol{z}})$ 为核函数。即核函数将输入空间中的任意两个向量 $\vec{\boldsymbol{x}},\vec{\boldsymbol{z}}$ 映射为特征空间中对应的向量之间的内积。

通常我们不关心这个映射的具体表达形式，而是直接给出 $K(\vec{\boldsymbol{x}},\vec{\boldsymbol{z}})$。

考虑到在线性支持向量机的对偶形式中，只涉及输入实例和实例之间的内积，故将内积 $\vec{\boldsymbol{x}}_i\cdot\vec{\boldsymbol{x}}_j$ 替换成核函数 $K(\vec{\boldsymbol{x}}_i,\vec{\boldsymbol{x}}_j)=\phi(\vec{\boldsymbol{x}}_i)\cdot\phi(\vec{\boldsymbol{x}}_j)$。则对偶问题的目标函数成为：

$$W(\vec{\alpha})=\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_jy_iy_jK(\vec{\boldsymbol{x}}_i,\vec{\boldsymbol{x}}_j)-\sum_{i=1}^{N}\alpha_i$$

分类决策函数变成：

$$f(\vec{\boldsymbol{x}})=\text{sign}(\sum_{i=1}^{N}\alpha_i^*y_iK(\vec{\boldsymbol{x}}_i,\vec{\boldsymbol{x}})+b^*)$$

在给定核函数 $K(\vec{\boldsymbol{x}},\vec{\boldsymbol{z}})$ 的情况下，可以利用求解线性分类问题的方法求解非线性分类问题的支持向量机。学习是隐式地在特征空间进行的，这样的技巧称为核技巧。在实际应用中，往往依赖经验直接选择核函数，然后验证该核函数确实是有效的核函数。给出常用的一些核函数如下。

- 多项式核函数：$K(\vec{\boldsymbol{x}},\vec{\boldsymbol{z}})=(\vec{\boldsymbol{x}}\cdot\vec{\boldsymbol{z}}+1)^p$
  - 对应的支持向量机是一个 p 次多项式分类器。
  - 此时分类决策函数成为

$$f(\vec{\boldsymbol{x}})=\text{sign}(\sum_{i=1}^{N}\alpha_i^*y_i(\vec{\boldsymbol{x}}_i\cdot\vec{\boldsymbol{x}}+1)^p+b^*)$$

- 高斯核函数：

$$K(\vec{\boldsymbol{x}},\vec{\boldsymbol{z}})=\exp(-\frac{||\boldsymbol{x}-\boldsymbol{z}||_2^2}{2\sigma^2})$$

  - 对应的支持向量机是高斯径向基函数分类器 (`radial basis function`)。
  - 此时的分类决策函数成为

$$f(\vec{\boldsymbol{x}})=\text{sign}(\sum_{i=1}^{N}\alpha_i^*y_i\exp(-\frac{||\boldsymbol{x_i}-\boldsymbol{x}||_2^2}{2\sigma^2})+b^*)$$

❑ sigmoid核函数：$K(\vec{x},\vec{z})=\tanh(\gamma(\vec{x}\cdot\vec{z})+r)$。

最后我们总结非线性支持向量机学习算法。

❑ 输入
 ❍ 训练数据集 $T=\{(\vec{x}_1,y_1),(\vec{x}_2,y_2),\cdots,(\vec{x}_N,y_N)\}$。
 ❍ 惩罚参数 $C>0$。

❑ 输出：分类决策函数。

❑ 算法步骤如下。
 ❍ 选择适当的核函数 $K(\vec{x},\vec{z})$，求解约束最优化问题：

$$\min_{\vec{\alpha}} \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j K(\vec{x}_i,\vec{x}_j)-\sum_{i=1}^{N}\alpha_i$$

$$s.t.\quad \sum_{i=1}^{N}\alpha_i y_i=0$$

$$C\geqslant\alpha_i\geqslant 0, i=1,2,\cdots,N$$

 求得最优解 $\vec{\alpha}^*=(\alpha_1^*,\alpha_2^*,\cdots,\alpha_N^*)^T$。

 ❍ 计算

$$\vec{w}^*=\sum_{i=1}^{N}\alpha_i^* y_i\vec{x}_i$$

 同时选择 $\vec{\alpha}^*$ 的某个合适的分量 $C>\alpha_j^*>0$，计算

$$b^*=y_j-\sum_{i=1}^{N}\alpha_i^* y_i K(\vec{x}_i,\vec{x}_j)$$

 ❍ 构造分类决策函数 $f(\vec{x})=\mathrm{sign}(\sum_{i=1}^{N}\alpha_i^* y_i K(\vec{x}_i,\vec{x})+b^*)$。

> 当 $K(\vec{x},\vec{z})$ 是正定核函数时，该问题为凸二次规划问题，解是存在的。

7.2.4 支持向量回归

支持向量机不仅可以用于分类问题，也可以用于回归问题。给定训练数据集 $T=\{(\vec{x}_1,y_1),(\vec{x}_2,y_2),\cdots,(\vec{x}_N,y_N)\}$，其中 $\vec{x}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(n)})^T\in\mathcal{X}=\mathbb{R}^n, y_i\in\mathcal{Y}=\mathbb{R}, i=1,2,\cdots,N$。对于样本 $(\vec{x}_i,y_i)$ 通常根据模型输出 $f(\vec{x}_i)$ 与真实值 y_i 之间的差别来计算损失，当且仅当 $f(\vec{x}_i)=y_i$ 时损失才为零。

支持向量回归 (Support Vector Regression，SVR) 的基本思路是：允许 $f(\vec{\boldsymbol{x}}_i)$ 与 y_i 之间最多有 ϵ 的偏差。仅当 $|f(\vec{\boldsymbol{x}}_i)-y_i|>\epsilon$ 时，才计算损失。当 $|f(\vec{\boldsymbol{x}}_i)-y_i|\leqslant\epsilon$ 时，我们认为预测正确。

用数学语言描述SVR问题：

$$\min_{\vec{\boldsymbol{w}},b}\frac{1}{2}||\vec{\boldsymbol{w}}||_2^2+C\sum_{i=1}^{N}L_\epsilon\left(f(\vec{\boldsymbol{x}}_i)-y_i\right)$$

其中 $C\geqslant 0$ 为罚项常数，L_ϵ 为损失函数 L_ϵ，如图 7.2 所示。L_ϵ 定义为：

$$L_\epsilon(z)=\begin{cases}0 & ,\text{if}|z|\leqslant\epsilon\\ |z|-\xi & ,\text{else}\end{cases}$$

线性回归中，损失函数为 $L(z)=z^2$。

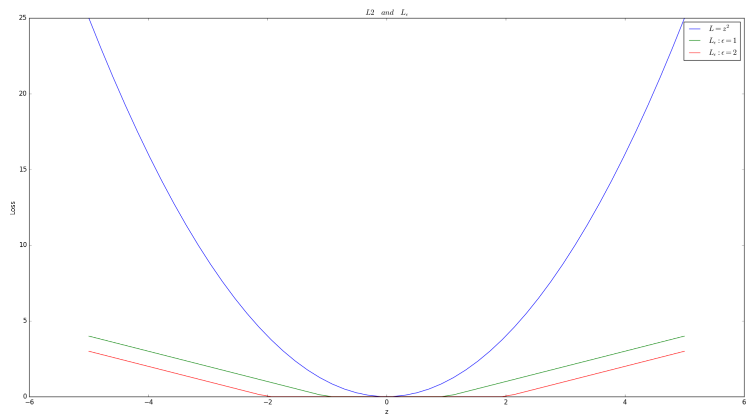


图 7.2 L_epsilon

更进一步，引入松弛变量 $\xi_i,\hat{\xi}_i$，则新的最优化问题为：

$$\begin{aligned}&\min_{\vec{\boldsymbol{w}},b,\xi_i,\hat{\xi}_i}\frac{1}{2}||\vec{\boldsymbol{w}}||_2^2+C\sum_{i=1}^{N}(\xi_i+\hat{\xi}_i)\\&s.t. f(\vec{\boldsymbol{x}}_i)-y_i\leqslant\epsilon+\xi_i,\\&y_i-f(\vec{\boldsymbol{x}}_i)\leqslant\epsilon+\hat{\xi}_i,\\&\xi_i\geqslant 0,\hat{\xi}_i\geqslant 0,i=1,2,\cdots,N\end{aligned}$$

这就是 SVR原始问题。类似地，引入拉格朗日乘子，$\mu_i \geqslant 0, \hat{\mu}_i \geqslant 0, \alpha_i \geqslant 0, \hat{\alpha}_i \geqslant 0$，定义拉格朗日函数：

$$
\begin{aligned}
L(\vec{\boldsymbol{w}}, b, \vec{\alpha}, \hat{\vec{\alpha}}, \vec{\xi}, \hat{\vec{\xi}}, \vec{\mu}, \hat{\vec{\mu}}) = \frac{1}{2}||\vec{\boldsymbol{w}}||_2^2 + C\sum_{i=1}^{N}(\xi_i + \hat{\xi}_i) - \sum_{i=1}^{N}\mu_i\xi_i - \sum_{i-1}^{N}\hat{\mu}_i\hat{\xi}_i \\
+ \sum_{i=1}^{N}\alpha_i\left(f(\vec{\boldsymbol{x}}_i) - y_i - \epsilon - \xi_i\right) + \sum_{i-1}^{N}\hat{\alpha}_i\left(y_i - f(\vec{\boldsymbol{x}}_i) - \epsilon - \hat{\xi}_i\right)
\end{aligned}
$$

根据拉格朗日对偶性，原始问题的对偶问题是极大极小问题

$$
\max_{\vec{\alpha}, \hat{\vec{\alpha}}} \min_{\vec{\boldsymbol{w}}, b, \vec{\xi}, \hat{\vec{\xi}}} L(\vec{\boldsymbol{w}}, b, \vec{\alpha}, \hat{\vec{\alpha}}, \vec{\xi}, \hat{\vec{\xi}}, \vec{\mu}, \hat{\vec{\mu}})
$$

先求极小问题：根据 $L(\vec{\boldsymbol{w}}, b, \vec{\alpha}, \hat{\vec{\alpha}}, \vec{\xi}, \hat{\vec{\xi}}, \vec{\mu}, \hat{\vec{\mu}})$ 对 $\vec{\boldsymbol{w}}, b, \vec{\xi}, \hat{\vec{\xi}}$ 偏导数为零可得：

$$
\begin{aligned}
\vec{\boldsymbol{w}} &= \sum_{i=1}^{N}(\hat{\alpha}_i - \alpha_i)\vec{\boldsymbol{x}}_i \\
0 &= \sum_{i=1}^{N}(\hat{\alpha}_i - \alpha_i) \\
C &= \alpha_i + \mu_i \\
C &= \hat{\alpha}_i + \hat{\mu}_i
\end{aligned}
$$

再求极大问题（取负号变极小问题）：

$$
\begin{aligned}
\min_{\vec{\alpha}, \hat{\vec{\alpha}}} \sum_{i=1}^{N}\left[y_i(\hat{\alpha}_i - \alpha_i) - \epsilon(\hat{\alpha}_i + \alpha_i)\right] - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}(\hat{\alpha}_i - \alpha_i)(\hat{\alpha}_j - \alpha_j)\vec{\boldsymbol{x}}_i^T\vec{\boldsymbol{x}}_j \\
s.t. \sum_{i=1}^{N}(\hat{\alpha}_i - \alpha_i) = 0 \\
0 \leqslant \alpha_i, \hat{\alpha}_i \leqslant C
\end{aligned}
$$

KKT 条件为：

$$
\begin{cases}
\alpha_i\left(f(\vec{\boldsymbol{x}}_i) - y_i - \epsilon - \xi_i\right) = 0 \\
\hat{\alpha}_i\left(y_i - f(\vec{\boldsymbol{x}}_i) - \epsilon - \hat{\xi}_i\right) = 0 \\
\alpha_i\hat{\alpha}_i = 0 \\
\xi_i\hat{\xi}_i = 0 \\
(C - \alpha_i)\xi_i = 0 \\
(C - \hat{\alpha}_i)\hat{\xi}_i = 0
\end{cases}
$$

> 由 KKT 条件可得：
> 当且仅当 $f(\vec{\boldsymbol{x}}_i) - y_i - \epsilon - \xi_i = 0$ 时，α_i 非零；
> 当且仅当 $y_i - f(\vec{\boldsymbol{x}}_i) - \epsilon - \hat{\xi}_i = 0$ 时，$\hat{\alpha}_i$ 非零。
> 约束 $f(\vec{\boldsymbol{x}}_i) - y_i - \epsilon - \xi_i = 0$ 与 $y_i - f(\vec{\boldsymbol{x}}_i) - \epsilon - \hat{\xi}_i = 0$ 不能同时成立（若同时成立，则得出 $\xi_i = 0, \hat{\xi}_i = 0$），因此 $\alpha_i, \hat{\alpha}_i$ 中至少一个为零。

假设最终解为 $\vec{\alpha}^* = (\alpha_1^*, \alpha_2^*, \cdots, \alpha_N^*)^T$，在 $\hat{\vec{\alpha}}^* = (\hat{\alpha}_1^*, \hat{\alpha}_2^*, \cdots, \hat{\alpha}_N^*)^T$ 中，找出 $\vec{\alpha}^*$ 的某个分量 $C > \alpha_j^* > 0$，则有：

$$b^* = y_j + \epsilon - \sum_{i=1}^{N} (\hat{\alpha}_i^* - \alpha_j^*) \vec{\boldsymbol{x}}_i^T \vec{\boldsymbol{x}}_j$$

$$f(\vec{\boldsymbol{x}}) = \sum_{i=1}^{N} (\hat{\alpha}_i^* - \alpha_i^*) \vec{\boldsymbol{x}}_i^T \vec{\boldsymbol{x}} + b^*$$

更进一步，如果考虑使用核技巧，给定核函数 $K(\vec{\boldsymbol{x}}_i, \vec{\boldsymbol{x}})$，则SVR可以表示为：

$$f(\vec{\boldsymbol{x}}) = \sum_{i=1}^{N} (\hat{\alpha}_i - \alpha_i) K(\vec{\boldsymbol{x}}_i, \vec{\boldsymbol{x}}) + b$$

7.2.5 SVM 的优缺点

支持向量机（Support Vector Machine，SVM）本质上是非线性方法，在样本量比较少的时候，容易抓住数据和特征之间的非线性关系（相比线性分类方法如 logistic regression），因此可以解决非线性问题、可以避免神经网络结构选择和局部极小点问题、可以提高泛化性能、可以解决高维问题。

SVM 对缺失数据敏感, 对非线性问题没有通用解决方案，必须谨慎选择核函数来处理，计算复杂度高。主流的算法是 $O(n^2)$，这样对大规模数据就显得很无力了。不仅如此，由于其存在两个对结果影响相当大的超参数（如果用 RBF 核，是核函数的参数 gamma 以及惩罚项 C），这两个超参数无法通过概率方法进行计算，只能通过穷举试验来求出，计算时间要远高于不少类似的非线性分类器。

7.3 Python 实战

首先导入包：

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets, linear_model,cross_validation,svm
```

在支持向量回归问题中，使用的数据集是scikit-learn自带的一个糖尿病病人的数据集。该数据集从糖尿病病人采样并整理后，特点如下：

- 数据集有 442 个样本；
- 每个样本有 10 个特征；
- 每个特征都是浮点数，数据都在 –0.2 ~ 0.2 之间；
- 样本的目标在整数 25 ~ 346 之间。

这里给出加载数据集的函数：

```
def load_data_regression():
    diabetes = datasets.load_diabetes()
    return cross_validation.train_test_split(diabetes.data,diabetes.target,
        test_size=0.25,random_state=0)
```

- 返回值：一个元组，元组依次是：训练样本集、测试样本集、训练样本集对应的标签值、测试样本集对应的标签值。

load_data_regression函数加载数据集并随机切分数据集为两个部分，其中test_size指定了测试集为原始数据集的大小（比例）。

在支持向量分类问题中，使用鸢尾花数据集。鸢尾花数据集一共有 150 个数据，这些数据分为 3 类（分别为setosa，versicolor，virginica），每类 50 个数据。每个数据包含 4 个属性：萼片 (sepal) 长度、萼片宽度、花瓣 (petal) 长度、花瓣宽度。

给出加载数据的函数：

```
def load_data_classfication():
    iris=datasets.load_iris()
    X_train=iris.data
    y_train=iris.target
    return cross_validation.train_test_split(X_train, y_train,test_size=0.25,
        random_state=0,stratify=y_train)
```

- 返回值：一个元组，元组依次是：训练样本集、测试样本集、训练样本集对应的标签值、测试样本集对应的标签值。

在这里采用分层采样。因为原始数据集中，前 50 个样本都是类别 0，中间 50 个样本都是类别 1，最后 50 个样本都是类别 2。如果不采取分层采用，那么最后切分得到的测试数据集就不是无偏的了。

7.3.1 线性分类 SVM

LinearSVC 实现了线性分类支持向量机，它是根据liblinear实现的，可以用于二类分类，也可以用于多类分类。其原型为：

```
sklearn.svm.LinearSVC(penalty='l2', loss='squared_hinge', dual=True, tol=0.0001, C=1.0,
 multi_class='ovr', fit_intercept=True, intercept_scaling=1, class_weight=None,
verbose=0, random_state=None, max_iter=1000)
```

其参数如下。

- C：一个浮点数，罚项参数。
- loss：字符串。表示损失函数。可以为如下。
  - 'hinge'：此时为合页损失函数（它是标准 SVM 的损失函数）；
  - 'squared_hinge'：合页损失函数的平方。
- penalty：字符串。指定 'l1'或者'l2'，罚项的范数。默认为 'l2'（它是标准 SVC 采用的）。
- dual：布尔值。如果为True，则解决对偶问题；如果是False，则解决原始问题。当n_samples > n_features时，倾向于采用False。
- tol：浮点数，指定终止迭代的阈值。
- multi_class: 字符串，指定多类分类问题的策略。
  - 'ovr'：采用one-vs-rest分类策略；
  - 'crammer_singer'：多类联合分类，很少用。因为它的计算量大，而且精度不会更佳，此时忽略loss,penalty,dual参数。
- fit_intercept：布尔值。如果为True，则计算截距，即决策函数中的常数项；否则忽略截距。
- intercept_scaling：浮点值。如果提供了，则实例 X变成向量 [X,intercept_scaling]。此时相当于添加了一个人工特征，该特征对所有实例都是常数值。
  - 此时截距变成：intercept_scaling* 人工特征的权重 w_s；
  - 此时人工特征也参与了罚项的计算。
- class_weight：可以是个字典，或者字符串'balanced'。指定各个类的权重，若未提供，则认为类的权重为 1。
  - 如果是字典，则指定每个类标签的权重；
  - 如果是'balanced'，则每个类的权重是它出现频率的倒数。
- verbose：一个整数，表示是否开启verbose输出。
- random_state：一个整数或者一个RandomState实例，或者None。
  - 如果为整数，则它指定随机数生成器的种子。
  - 如果为RandomState实例，则指定随机数生成器。
  - 如果为None，则使用默认的随机数生成器。
- max_iter：一个整数，指定最大的迭代次数。

其属性如下。

- coef_：一个数组，它给出了各个特征的权重。

- ❑ intercept_：一个数组，它给出了截距，即决策函数中的常数项。

其方法如下。

- ❑ fit(X, y)：训练模型。
- ❑ predict(X)：用模型进行预测，返回预测值。
- ❑ score(X,y[,sample_weight])：返回在 (X,y)上预测的准确率 (accuracy)。

使用LinearSVC类考察线性分类支持向量机的预测能力，给出函数：

```
def test_LinearSVC(*data):
    X_train,X_test,y_train,y_test=data
    cls=svm.LinearSVC()
    cls.fit(X_train,y_train)
    print('Coefficients:%s, intercept %s'%(cls.coef_,cls.intercept_))
    print('Score: %.2f' % cls.score(X_test, y_test))
```

调用test_LinearSVC函数：

```
X_train,X_test,y_train,y_test=load_data_classfication()
test_LinearSVC(X_train,X_test,y_train,y_test)
```

输出如下：

```
Coefficients:[[ 0.21663419  0.38734345 -0.82116781 -0.44244686]
 [-0.14334854 -0.76758508  0.52335218 -1.00210446]
 [-0.80600556 -0.91883527  1.25796797  1.7256806 ]],
 intercept [ 0.12376265  2.03555561 -1.40078178]
Score: 0.97
```

可以看到线性分类支持向量机的预测性能相当好，对于测试集的预测准确率高达 97%。

考察损失函数的影响，这里给出函数：

```
def test_LinearSVC_loss(*data):
    X_train,X_test,y_train,y_test=data
    losses=['hinge','squared_hinge']
    for loss in losses:
        cls=svm.LinearSVC(loss=loss)
        cls.fit(X_train,y_train)
        print("Loss:%f"%loss)
        print('Coefficients:%s, intercept %s'%(cls.coef_,cls.intercept_))
        print('Score: %.2f' % cls.score(X_test, y_test))
```

类似test_LinearSVC函数的调用方式，得到输出如下，可以看到在鸢尾花分类这个问题上，虽然支持向量机的损失函数不同，但是它们对于测试集的预测准确率都相同。

```
Loss:hinge
Coefficients:[[ 0.38614299  0.28053909 -1.07892009 -0.5668941 ]
```

```
 [ 0.47015078 -1.55587825  0.40339917 -1.35402184]
 [-1.20338368 -1.18808419  1.87538952  1.85657256]],
intercept [ 0.18978095  1.34245263 -1.35134744]
Score: 0.97
Loss:squared_hinge
Coefficients:[[ 0.21663019  0.38734214 -0.82116655 -0.44244521]
 [-0.14099397 -0.7669855   0.52075121 -1.00018766]
 [-0.80599382 -0.91889119  1.25797774  1.72582122]],
 intercept [ 0.12375962  2.03014569 -1.40070379]
Score: 0.97
```

接着考察罚项形式的影响，这里给出函数：

```
def test_LinearSVC_L12(*data):
    X_train,X_test,y_train,y_test=data
    L12=['l1','l2']
    for p in L12:
        cls=svm.LinearSVC(penalty=p,dual=False)
        cls.fit(X_train,y_train)
        print("penalty:%s"%p)
        print('Coefficients:%s, intercept %s'%(cls.coef_,cls.intercept_))
        print('Score: %.2f' % cls.score(X_test, y_test))
```

这里dual=False是因为当dual=True，penalty=l2的情况不支持。类似test_LinearSVC函数的调用方式，得到输出如下，可以看到在鸢尾花分类这个问题上，虽然支持向量机的罚项形式不同，但是它们对于测试集的预测准确率都相同。

```
penalty:l1
Coefficients:[[ 0.16484821  0.52043853 -0.93367411  0.        ]
 [-0.16306709 -0.8894536   0.47984342 -0.90652583]
 [-0.58113822 -0.88218038  1.02755938  2.11411658]],
 intercept [ 0.          2.56482119 -2.41988516]
Score: 0.97
penalty:l2
Coefficients:[[ 0.22265159  0.38178558 -0.82875399 -0.43752314]
 [-0.14420829 -0.76854603  0.52438222 -1.0031107 ]
 [-0.80624982 -0.91903181  1.25881817  1.72510131]],
intercept [ 0.12290274  2.03987493 -1.40231716]
Score: 0.97
```

最后考察罚项系数C的影响。C衡量了误分类点的重要性，C越大则误分类点越重要。这里给出函数：

```
def test_LinearSVC_C(*data):
    X_train,X_test,y_train,y_test=data
    Cs=np.logspace(-2,1)
    train_scores=[]
```

```
test_scores=[]
for C in Cs:
    cls=svm.LinearSVC(C=C)
    cls.fit(X_train,y_train)
    train_scores.append(cls.score(X_train,y_train))
    test_scores.append(cls.score(X_test,y_test))

## 绘图
fig=plt.figure()
ax=fig.add_subplot(1,1,1)
ax.plot(Cs,train_scores,label="Traing score")
ax.plot(Cs,test_scores,label="Testing score")
ax.set_xlabel(r"C")
ax.set_ylabel(r"score")
ax.set_xscale('log')
ax.set_title("LinearSVC")
ax.legend(loc='best')
plt.show()
```

类似test_LinearSVC函数的调用方式，得到输出图形如图 7.3 所示。为了便于观察，将 x 轴以对数表示。可以看到当C较小时，误分类点重要性较低，此时误分类点较多，分类器性能较差。

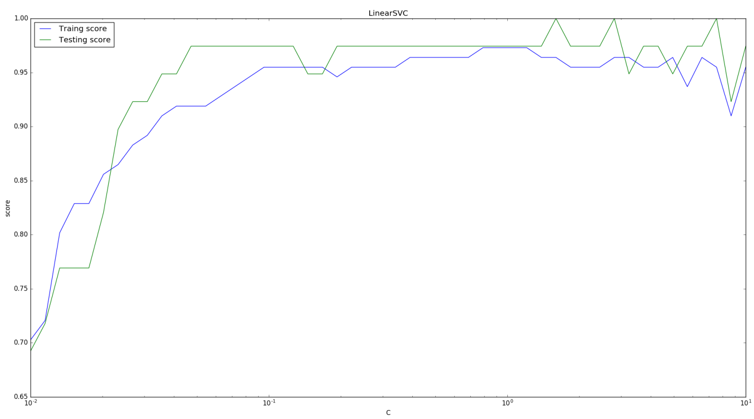


图 7.3 LinearSVC_C

7.3.2 非线性分类 SVM

SVC实现了非线性分类支持向量机，它是根据libsvm实现的，可以用于二类分类，也可以用于多类分类。其原型为：

```
sklearn.svm.SVC(C=1.0, kernel='rbf', degree=3, gamma='auto', coef0=0.0, shrinking=True,
 probability=False, tol=0.001, cache_size=200, class_weight=None, verbose=False,
 max_iter=-1, decision_function_shape=None, random_state=None)
```

SVC是根据libsvm实现的，是C-Support Vector Classfication。其训练的时间复杂度是采样点数量的平方。它可以用于二类分类，也可以用于多类分类问题（此时默认是根据one-vs-rest原则来分类的）。

参数如下。

- C：一个浮点数，罚项系数。
- kernel：一个字符串，指定核函数。
  - 'linear'：线性核 $K(\vec{x},\vec{z}) = \vec{x} \cdot \vec{z}$。
  - 'poly'：多项式核 $K(\vec{x},\vec{z}) = (\gamma(\vec{x} \cdot \vec{z} + 1) + r)^p$，其中 p 由 degree参数决定，γ 由 gamma参数决定，r 由 coef0参数决定。
  - 'rbf'（默认值）：高斯核函数 $K(\vec{x},\vec{z}) = \exp(-\gamma||\vec{x} - \vec{z}||^2)$，其中 γ 由 gamma参数决定。
  - 'sigmoid'：$K(\vec{x},\vec{z}) = \tanh(\gamma(\vec{x} \cdot \vec{z}) + r)$。其中 γ 由 gamma参数决定，r 由 coef0参数指定。
  - 'precomputed'：表示提供了kernel matrix，或者提供一个可调用对象，该对象用于计算kernel matrix。
- degree：一个整数。指定当核函数是多项式核函数时，多项式的系数。对于其他核函数，该参数无效。
- gamma：一个浮点数。当核函数是'rbf'，'poly'，'sigmoid'时，核函数的系数。如果为'auto'，则表示系数为1/n_features。
- coef0：浮点数，用于指定核函数中的自由项。只有当核函数是'poly'和'sigmoid'时有效。
- probability：布尔值。如果为True，则会进行概率估计。它必须在训练之前设置好，且概率估计会拖慢训练速度。
- shrinking：布尔值。如果为True，则使用启发式收缩(shrinking heuristic)。
- tol：浮点数，指定终止迭代的阈值。
- cache_size：浮点值，指定了kernel cache的大小，单位为 MB。
- class_weight：可以是个字典，或者字符串'balanced'。指定各个类的权重，若未提供，则认为类的权重为 1。
  - 如果是字典，则指定每个类标签的权重；
  - 如果是'balanced'，则每个类的权重是它出现频数的倒数。
- verbose：一个整数，表示是否开启verbose输出。
- max_iter：一个整数，指定最大迭代次数。
- decision_function_shape：为字符串或者None，指定决策函数的形状。
  - 'ovr'：则使用one-vs-rest准则，那么决策函数形状是(n_samples,n_classes)。

> 此时对每个分类定义了一个二类 SVM，一共 n_classes个二类 SVM 组合成一个多类 SVM。

- 'ovo'：则使用one-vs-one准测，那么决策函数形状是 (n_samples, n_classes * (n_classes - 1) / 2)。

> 此时对每一对分类直接定义了一个二类 SVM，一共 n_classes * (n_classes - 1) / 2)个二类 SVM 组合成一个多类 SVM。

- None：默认值。采用该值时，目前会使用'ovo'，但是在 scikit v0.18之后切换成'ovr'。

- random_state：一个整数，或者一个RandomState实例，或者None。
  - 如果为整数，则它指定随机数生成器的种子。
  - 如果为RandomState实例，则指定随机数生成器。
  - 如果为None，则使用默认的随机数生成器。

属性如下。

- support_：一个数组，形状为 [n_SV]，支持向量的下标。
- support_vectors_：一个数组，形状为 [n_SV, n_features]，支持向量。
- n_support_：一个数组-like，形状为 [n_class]. 每一个分类的支持向量的个数。
- dual_coef_：一个数组，形状为[n_class-1, n_SV]。对偶问题中，在分类决策函数中每个支持向量的系数。
- coef_：一个数组，形状为[n_class-1, n_features]。原始问题中，每个特征的系数。只有在linear kernel中有效。

> coef_ 是个只读的属性。它是从dual_coef_ 和support_vectors_计算而来的。

- intercept_：一个数组，形状为[n_class * (n_class-1) / 2]，决策函数中的常数项。

方法如下。

- fit(X, y[, sample_weight])：训练模型。
- predict(X)：用模型进行预测，返回预测值。
- score(X,y[,sample_weight])：返回在 (X,y)上预测的准确率 (accuracy)。
- predict_log_proba(X)：返回一个数组，数组的元素依次是X预测为各个类别的概率的对数值。
- predict_proba(X)：返回一个数组，数组的元素依次是X预测为各个类别的概率值。

这里主要观察不同的核对预测性能的影响。首先观察最简单的线性核 $K(\vec{x},\vec{z}) = \vec{x}\cdot\vec{z}$，给出测试函数如下：

```
def test_SVC_linear(*data):
    X_train,X_test,y_train,y_test=data
    cls=svm.SVC(kernel='linear')
    cls.fit(X_train,y_train)
    print('Coefficients:%s, intercept %s'%(cls.coef_,cls.intercept_))
    print('Score: %.2f' % cls.score(X_test, y_test))
```

类似test_LinearSVC函数的调用方式，得到输出如下：

```
Coefficients:[[-0.09678346  0.40464239 -0.95269008 -0.50457359]
 [ 0.01882639  0.17448893 -0.54596525 -0.21811023]
 [ 0.487078    0.93180889 -1.77348523 -1.99541406]],
intercept [ 1.89493378  1.29943173  6.38471237]
Score: 1.00
```

可以看到线性核要比线性分类支持向量机LinearSVC的预测效果更佳，对测试集的预测全部正确。

接下来考察多项式核：$K(\vec{x},\vec{z})=(\gamma(\vec{x}\cdot\vec{z}+1)+r)^p$，其中 p 由 degree参数决定，γ 由 gamma参数决定，r 由 coef0参数决定。给出测试函数如下：

```
def test_SVC_poly(*data):
    X_train,X_test,y_train,y_test=data
    fig=plt.figure()
    ### 测试 degree ####
    degrees=range(1,20)
    train_scores=[]
    test_scores=[]
    for degree in degrees:
        cls=svm.SVC(kernel='poly',degree=degree)
        cls.fit(X_train,y_train)
        train_scores.append(cls.score(X_train,y_train))
        test_scores.append(cls.score(X_test, y_test))
    ax=fig.add_subplot(1,3,1)
    ax.plot(degrees,train_scores,label="Training score ",marker='+' )
    ax.plot(degrees,test_scores,label= " Testing  score ",marker='o' )
    ax.set_title( "SVC_poly_degree ")
    ax.set_xlabel("p")
    ax.set_ylabel("score")
    ax.set_ylim(0,1.05)
    ax.legend(loc="best",framealpha=0.5)

    ### 测试 gamma ####
    gammas=range(1,20)
    train_scores=[]
    test_scores=[]
    for gamma in gammas:
```

```
        cls=svm.SVC(kernel='poly',gamma=gamma,degree=3)
        cls.fit(X_train,y_train)
        train_scores.append(cls.score(X_train,y_train))
        test_scores.append(cls.score(X_test, y_test))
    ax=fig.add_subplot(1,3,2)
    ax.plot(gammas,train_scores,label="Training score ",marker='+' )
    ax.plot(gammas,test_scores,label= " Testing  score ",marker='o' )
    ax.set_title( "SVC_poly_gamma ")
    ax.set_xlabel(r"$\gamma$")
    ax.set_ylabel("score")
    ax.set_ylim(0,1.05)
    ax.legend(loc="best",framealpha=0.5)
    ### 测试 r ######
    rs=range(0,20)
    train_scores=[]
    test_scores=[]
    for r in rs:
        cls=svm.SVC(kernel='poly',gamma=10,degree=3,coef0=r)
        cls.fit(X_train,y_train)
        train_scores.append(cls.score(X_train,y_train))
        test_scores.append(cls.score(X_test, y_test))
    ax=fig.add_subplot(1,3,3)
    ax.plot(rs,train_scores,label="Training score ",marker='+' )
    ax.plot(rs,test_scores,label= " Testing  score ",marker='o' )
    ax.set_title( "SVC_poly_r ")
    ax.set_xlabel(r"r")
    ax.set_ylabel("score")
    ax.set_ylim(0,1.05)
    ax.legend(loc="best",framealpha=0.5)
    plt.show()
```

测试结果如图 7.4 所示。可以看到在测试集上的预测性能随 p 的变化比较平稳，随 γ 影响不大，在 $r=0$ 时性能最佳。

接下来考察高斯核：$K(\vec{\boldsymbol{x}},\vec{\boldsymbol{z}})=\exp(-\gamma||\vec{\boldsymbol{x}}-\vec{\boldsymbol{z}}||^2)$，其中 γ 由 gamma参数决定，给出测试函数如下：

```
def test_SVC_rbf(*data):
    X_train,X_test,y_train,y_test=data
    gammas=range(1,20)
    train_scores=[]
    test_scores=[]
    for gamma in gammas:
        cls=svm.SVC(kernel='rbf',gamma=gamma)
        cls.fit(X_train,y_train)
        train_scores.append(cls.score(X_train,y_train))
```

```
        test_scores.append(cls.score(X_test, y_test))
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    ax.plot(gammas,train_scores,label="Training score ",marker='+' )
    ax.plot(gammas,test_scores,label= " Testing  score ",marker='o' )
    ax.set_title( "SVC_rbf")
    ax.set_xlabel(r"$\gamma$")
    ax.set_ylabel("score")
    ax.set_ylim(0,1.05)
    ax.legend(loc="best",framealpha=0.5)
    plt.show()
```

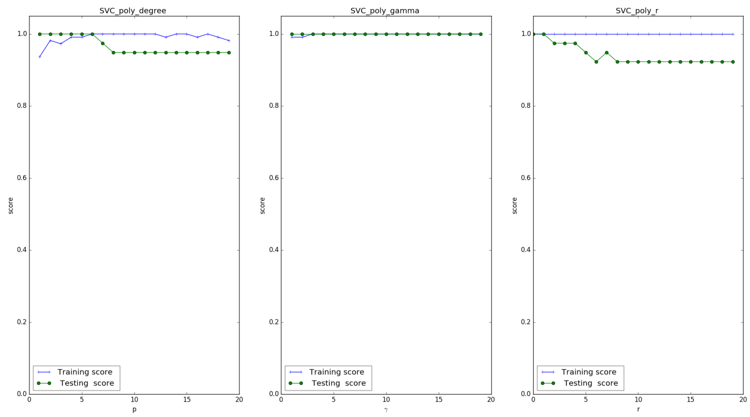


图 7.4 SVC_poly

测试结果如图 7.5 所示。可以看到在测试集上的预测性能随 γ 的变化比较平稳。

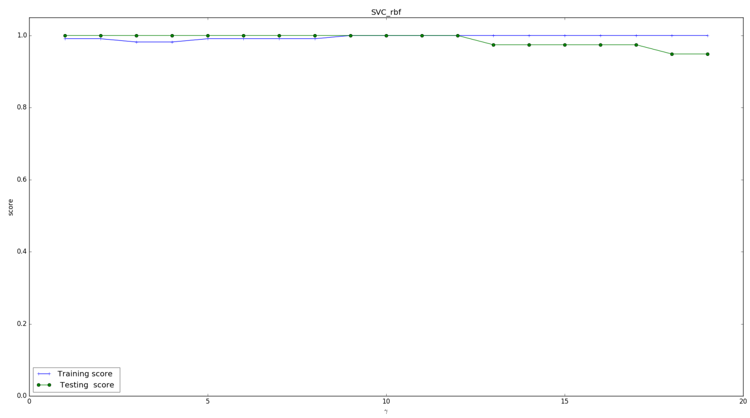


图 7.5 SVC_rbf

最后观察sigmoid核：$K(\vec{x},\vec{z}) = \tanh(\gamma(\vec{x}\cdot\vec{z})+r)$。其中 γ 由 gamma参数决定，r 由 coef0参数指定。给出测试函数如下：

```
def test_SVC_sigmod(*data):
    X_train,X_test,y_train,y_test=data
    fig=plt.figure()

    ### 测试 gamma ####
    gammas=np.logspace(-2,1)
    train_scores=[]
    test_scores=[]

    for gamma in gammas:
        cls=svm.SVC(kernel='sigmoid',gamma=gamma,coef0=0)
        cls.fit(X_train,y_train)
        train_scores.append(cls.score(X_train,y_train))
        test_scores.append(cls.score(X_test, y_test))
    ax=fig.add_subplot(1,2,1)
    ax.plot(gammas,train_scores,label="Training score ",marker='+' )
    ax.plot(gammas,test_scores,label= " Testing  score ",marker='o' )
    ax.set_title( "SVC_sigmoid_gamma ")
    ax.set_xscale("log")
    ax.set_xlabel(r"$\gamma$")
    ax.set_ylabel("score")
    ax.set_ylim(0,1.05)
    ax.legend(loc="best",framealpha=0.5)
    ### 测试 r ######
    rs=np.linspace(0,5)
    train_scores=[]
    test_scores=[]

    for r in rs:
        cls=svm.SVC(kernel='sigmoid',coef0=r,gamma=0.01)
        cls.fit(X_train,y_train)
        train_scores.append(cls.score(X_train,y_train))
        test_scores.append(cls.score(X_test, y_test))
    ax=fig.add_subplot(1,2,2)
    ax.plot(rs,train_scores,label="Training score ",marker='+' )
    ax.plot(rs,test_scores,label= " Testing  score ",marker='o' )
    ax.set_title( "SVC_sigmoid_r ")
    ax.set_xlabel(r"r")
    ax.set_ylabel("score")
    ax.set_ylim(0,1.05)
    ax.legend(loc="best",framealpha=0.5)
    plt.show()
```

测试结果如图 7.6 所示。可以看到在测试集上的预测都比较糟糕。固定 $r = 0$，预测性能随着 γ 的增长而下降；固定 $\gamma = 0.01$，预测性能随着 r 的增长而下降。

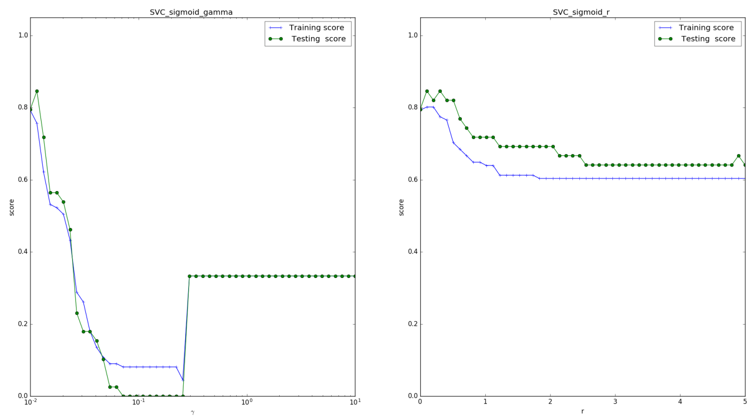


图 7.6 SVC_sigmoid

7.3.3 线性回归 SVR

LinearSVR 实现了线性回归支持向量机，它基于liblinear实现，其原型为：

```
class sklearn.svm.LinearSVR(epsilon=0.0, tol=0.0001, C=1.0, loss='epsilon_insensitive',
 fit_intercept=True, intercept_scaling=1.0, dual=True, verbose=0, random_state=None,
 max_iter=1000)
```

参数如下。

- C：一个浮点数，罚项系数。
- loss：字符串，表示损失函数。可以为
  - 'epsilon_insensitive'：此时损失函数为 L_ϵ （标准的SVR ）。
  - 'squared_epsilon_insensitive'：此时损失函数为 L_ϵ^2。
- epsilon：浮点数，用于lose中的 ϵ 参数。
- dual：布尔值。如果为True，则解决对偶问题；如果是False，则解决原始问题。当n_samples > n_features时，倾向于采用False。
- tol：浮点数，指定终止迭代的阈值。
- fit_intercept：布尔值。如果为True，则计算截距，即决策函数中的常数项；否则忽略截距。
- intercept_scaling：浮点值。如果提供了，则实例 X变成向量 [X,intercept_scaling]。此时相当于添加了一个人工特征，该特征对所有实例都是常数值。
  - 此时截距变成：intercept_scaling* 人工特征的权重 w_s。
  - 此时人工特征也参与了罚项的计算。
- verbose：一个整数，表示是否开启verbose输出。
- random_state：一个整数，或者一个RandomState实例，或者None。
  - 如果为整数，则它指定随机数生成器的种子；

- ❍ 如果为RandomState实例，则指定随机数生成器；
- ❍ 如果为None，则使用默认的随机数生成器。

❑ max_iter：一个整数，指定最大迭代次数。

属性如下。

❑ coef_：一个数组，它给出了各个特征的权重；

❑ intercept_：一个数组，它给出了截距，即决策函数中的常数项。

方法如下。

❑ fit(X, y)：训练模型。

❑ predict(X)：用模型进行预测，返回预测值。

❑ score(X,y[,sample_weight])：返回预测性能得分。设预测集为 T_{test}，真实值为 y_i，真实值的均值为 $\bar{y}$，预测值为 $\hat{y}_i$

$$score = 1 - \frac{\sum_{T_{test}}(y_i - \hat{y}_i)^2}{(y_i - \bar{y})^2}$$

- ❍ score不超过 1，但是可能为负值（预测效果太差）。
- ❍ score越大，预测性能越好。

使用LinearSVR类考察线性回归支持向量机的预测能力，给出函数：

```
def test_LinearSVR(*data):
    X_train,X_test,y_train,y_test=data
    regr=svm.LinearSVR()
    regr.fit(X_train,y_train)
    print('Coefficients:%s, intercept %s'%(regr.coef_,regr.intercept_))
    print('Score: %.2f' % regr.score(X_test, y_test))
```

调用test_LinearSVR函数：

```
X_train,X_test,y_train,y_test=load_data_regression()
test_LinearSVR(X_train,X_test,y_train,y_test)
```

输出如下，可以看到线性回归支持向量机的预测性能较差，对于测试集的预测得分仅为 −0.56。

```
Coefficients:[ 2.14940259  0.4418875   6.35258779  4.62357282  2.82085901  2.42005063
 -5.3367464   5.41765142  7.26812843  4.33778867], intercept [ 99.]
Score: -0.56
```

接下来考察损失函数类型对于预测性能的影响，给出测试函数：

```
def test_LinearSVR_loss(*data):
    X_train,X_test,y_train,y_test=data
```

```
    losses=['epsilon_insensitive','squared_epsilon_insensitive']
    for loss in losses:
        regr=svm.LinearSVR(loss=loss)
        regr.fit(X_train,y_train)
        print("loss: %s"%loss)
        print('Coefficients:%s, intercept %s'%(regr.coef_,regr.intercept_))
        print('Score: %.2f' % regr.score(X_test, y_test))
```

类似调用test_LinearSVR函数的方式调用test_LinearSVR_loss函数，结果如下：

```
loss: epsilon_insensitive
Coefficients:[ 2.14940259  0.4418875   6.35258779  4.62357282  2.82085901  2.42005063
 -5.3367464   5.41765142  7.26812843  4.33778867], intercept [ 99.]
Score: -0.56
loss: squared_epsilon_insensitive
Coefficients:[   7.05589254 -103.32702827  395.67718876  221.76267721  -11.07980198
  -63.55478403 -176.68102488  117.56455915  322.6374064    95.61796343],
intercept [ 152.37666327]
Score: 0.38
```

可以看到，当loss='squared_epsilon_insensitive'时预测性能更好。

接下来考察 ϵ 对于预测性能的影响。ϵ 代表只要预测值落在标准值附近 ϵ 宽的区间内，都标记为预测正确，给出测试函数：

```
def test_LinearSVR_epsilon(*data):
    X_train,X_test,y_train,y_test=data
    epsilons=np.logspace(-2,2)
    train_scores=[]
    test_scores=[]
    for  epsilon in  epsilons:
        regr=svm.LinearSVR(epsilon=epsilon,loss='squared_epsilon_insensitive')
        regr.fit(X_train,y_train)
        train_scores.append(regr.score(X_train, y_train))
        test_scores.append(regr.score(X_test, y_test))
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    ax.plot(epsilons,train_scores,label="Training score ",marker='+' )
    ax.plot(epsilons,test_scores,label= " Testing  score ",marker='o' )
    ax.set_title( "LinearSVR_epsilon ")
    ax.set_xscale("log")
    ax.set_xlabel(r"$\epsilon$")
    ax.set_ylabel("score")
    ax.set_ylim(-1,1.05)
    ax.legend(loc="best",framealpha=0.5)
    plt.show()
```

执行测试，结果如图 7.7 所示。为了方便观看，将 x 轴转换成对数坐标，可以看到预测准确率随着 ϵ 下降。

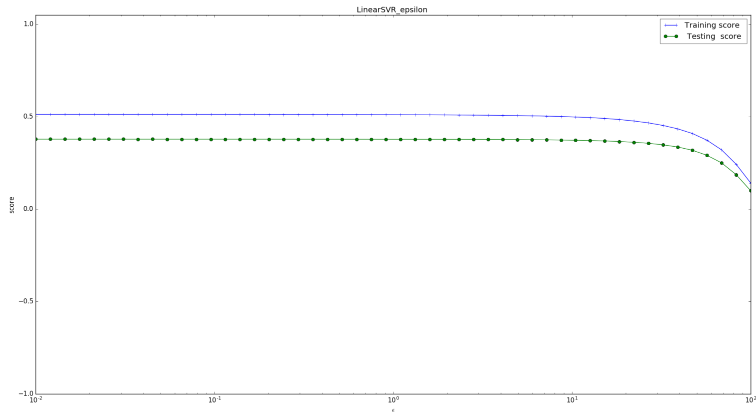


图 7.7 LinearSVR_epsilon

最后考察罚项系数C的影响。C衡量了误分类点的重要性，C越大则误分类点越重要。这里给出函数：

```
def test_LinearSVR_C(*data):
    X_train,X_test,y_train,y_test=data
    Cs=np.logspace(-1,2)
    train_scores=[]
    test_scores=[]
    for  C in  Cs:
        regr=svm.LinearSVR(epsilon=0.1,loss='squared_epsilon_insensitive',C=C)
        regr.fit(X_train,y_train)
        train_scores.append(regr.score(X_train, y_train))
        test_scores.append(regr.score(X_test, y_test))
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    ax.plot(Cs,train_scores,label="Training score ",marker='+' )
    ax.plot(Cs,test_scores,label= " Testing  score ",marker='o' )
    ax.set_title( "LinearSVR_C ")
    ax.set_xscale("log")
    ax.set_xlabel(r"C")
    ax.set_ylabel("score")
    ax.set_ylim(-1,1.05)
    ax.legend(loc="best",framealpha=0.5)
    plt.show()
```

执行测试，结果如图 7.8 所示。为了方便观看，将x轴转换成对数坐标。可以看到预测准确率随着 C 增大而上升。说明越看重误分类点，则预测得越准确。

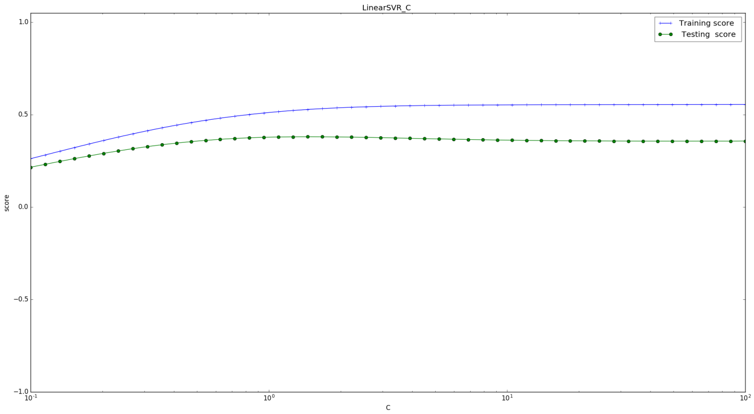


图 7.8 LinearSVR_C

7.3.4 非线性回归 SVR

SVR实现了非线性回归支持向量机，它是基于libsvm实现的，其原型为：

```
class sklearn.svm.SVR(kernel='rbf', degree=3, gamma='auto', coef0=0.0, tol=0.001, C=1.0,
 epsilon=0.1, shrinking=True, cache_size=200, verbose=False, max_iter=-1)
```

参数如下。

- C：一个浮点数，罚项系数。
- epsilon：一个浮点数，即 ϵ 参数。
- kernel：一个字符串，指定核函数。
  - 'linear'：线性核 $K(\vec{x},\vec{z}) = \boldsymbol{x}\cdot\boldsymbol{z}$。
  - 'poly'：多项式核 $K(\vec{x},\vec{z}) = (\gamma(\vec{x}\cdot\vec{z}+1)+r)^p$，其中 p 由 degree参数决定，γ 由 gamma参数决定，r 由 coef0参数决定。
  - 'rbf'（默认值）：高斯核函数 $K(\vec{x},\vec{z}) = \exp(-\gamma||\boldsymbol{x}-\boldsymbol{z}||^2)$，其中 γ 由 gamma参数决定。
  - 'sigmoid'：$K(\vec{x},\vec{z}) = \tanh(\gamma(\boldsymbol{x}\cdot\boldsymbol{z})+r)$，其中 γ 由 gamma参数决定，r 由 coef0参数指定。
  - 'precomputed'：表示提供了kernel matrix，
  - 或者提供一个可调用对象，该对象用于计算kernel matrix。
- degree：一个整数，指定当核函数是多项式核函数时，多项式的系数。对于其他核函数，该参数无效。
- gamma：一个浮点数，当核函数是'rbf'、'poly'、'sigmoid'时，核函数的系数。如果为'auto'，则表示系数为1/n_features。
- coef0：浮点数，用于指定核函数中的自由项。只有当核函数是'poly'和'sigmoid'时有效
- shrinking：布尔值。如果为True，则使用启发式(shrinking heuristic)。

- ❑ tol：浮点数，指定终止迭代的阈值。
- ❑ cache_size：浮点值，指定了kernel cache的大小，单位为 MB。
- ❑ verbose：一个整数，表示是否开启verbose输出。
- ❑ max_iter：一个整数，指定最大迭代次数。

属性如下。

- ❑ support_：一个数组，形状为 [n_SV]，支持向量的下标。
- ❑ support_vectors_：一个数组，形状为 [n_SV, n_features]，支持向量。
- ❑ n_support_：一个数组-like，形状为 [n_class]. 每一个分类的支持向量的个数。
- ❑ dual_coef_：一个数组，形状为[n_class-1, n_SV]，给出了决策函数中每个支持向量的系数。
- ❑ coef_：一个数组，形状为[n_class-1, n_features]。原始问题中每个特征的系数，只有在linear kernel中有效。

> coef_ 是个只读的属性。它是从dual_coef_ 和support_vectors_计算而来的。

- ❑ intercept_：决策函数中的常数项。

方法如下。

- ❑ fit(X, y)：训练模型。
- ❑ predict(X)：用模型进行预测，返回预测值。
- ❑ score(X,y[,sample_weight])：返回预测性能得分。设预测集为 T_{test}，真实值为 y_i，真实值的均值为 $\bar{y}$，预测值为 $\hat{y}_i$

$$score = 1 - \frac{\sum_{T_{test}}(y_i - \hat{y}_i)^2}{(y_i - \bar{y})^2}$$

  - ❍ score不超过 1，但是可能为负值（预测效果太差）;
  - ❍ score越大，预测性能越好。

这里主要观察不同的核对预测性能的影响。首先观察最简单的线性核 $K(\vec{\boldsymbol{x}},\vec{\boldsymbol{z}}) = \vec{\boldsymbol{x}} \cdot \vec{\boldsymbol{z}}$，给出测试函数如下：

```
def test_SVR_linear(*data):
    X_train,X_test,y_train,y_test=data
    regr=svm.SVR(kernel='linear')
    regr.fit(X_train,y_train)
    print('Coefficients:%s, intercept %s'%(regr.coef_,regr.intercept_))
    print('Score: %.2f' % regr.score(X_test, y_test))
```

类似test_LinearSVR函数的调用方式，得到输出如下。可以看到线性核要比线性回归支持向量机LinearSVR的预测效果（预测得分为 −0.56）更佳。

```
Coefficients:[[ 2.24127622 -0.38128702  7.87018376  5.21135861  2.26619436  1.70869458
  -5.7746489   5.51487251  7.94860817  4.59359657]], intercept [ 137.11012796]
Score: -0.03
```

接下来考察多项式核：$K(\vec{x},\vec{z}) = (\gamma(\vec{x}\cdot\vec{z}+1)+r)^p$，其中 p 由 degree参数决定，γ 由 gamma参数决定，r 由 coef0参数决定。给出测试函数如下：

```
def test_SVR_poly(*data):
    X_train,X_test,y_train,y_test=data
    fig=plt.figure()
    ### 测试 degree ####
    degrees=range(1,20)
    train_scores=[]
    test_scores=[]
    for degree in degrees:
        regr=svm.SVR(kernel='poly',degree=degree,coef0=1)
        regr.fit(X_train,y_train)
        train_scores.append(regr.score(X_train,y_train))
        test_scores.append(regr.score(X_test, y_test))
    ax=fig.add_subplot(1,3,1)
    ax.plot(degrees,train_scores,label="Training score ",marker='+' )
    ax.plot(degrees,test_scores,label= " Testing  score ",marker='o' )
    ax.set_title( "SVR_poly_degree r=1")
    ax.set_xlabel("p")
    ax.set_ylabel("score")
    ax.set_ylim(-1,1.)
    ax.legend(loc="best",framealpha=0.5)

    ### 测试 gamma ####
    gammas=range(1,40)
    train_scores=[]
    test_scores=[]
    for gamma in gammas:
        regr=svm.SVR(kernel='poly',gamma=gamma,degree=3,coef0=1)
        regr.fit(X_train,y_train)
        train_scores.append(regr.score(X_train,y_train))
        test_scores.append(regr.score(X_test, y_test))
    ax=fig.add_subplot(1,3,2)
    ax.plot(gammas,train_scores,label="Training score ",marker='+' )
    ax.plot(gammas,test_scores,label= " Testing  score ",marker='o' )
    ax.set_title( "SVR_poly_gamma  r=1")
    ax.set_xlabel(r"$\gamma$")
    ax.set_ylabel("score")
    ax.set_ylim(-1,1)
    ax.legend(loc="best",framealpha=0.5)
    ### 测试 r ######
```

```
    rs=range(0,20)
    train_scores=[]
    test_scores=[]
    for r in rs:
        regr=svm.SVR(kernel='poly',gamma=20,degree=3,coef0=r)
        regr.fit(X_train,y_train)
        train_scores.append(regr.score(X_train,y_train))
        test_scores.append(regr.score(X_test, y_test))
    ax=fig.add_subplot(1,3,3)
    ax.plot(rs,train_scores,label="Training score ",marker='+' )
    ax.plot(rs,test_scores,label= " Testing  score ",marker='o' )
    ax.set_title( "SVR_poly_r gamma=20 degree=3")
    ax.set_xlabel(r"r")
    ax.set_ylabel("score")
    ax.set_ylim(-1,1.)
    ax.legend(loc="best",framealpha=0.5)
    plt.show()
```

测试结果如图 7.9 所示。可以看到在测试集上的预测性能随 p 的变化比较平稳，随 γ 增大而增大，随 $r=0$ 增大时先增大后平稳。

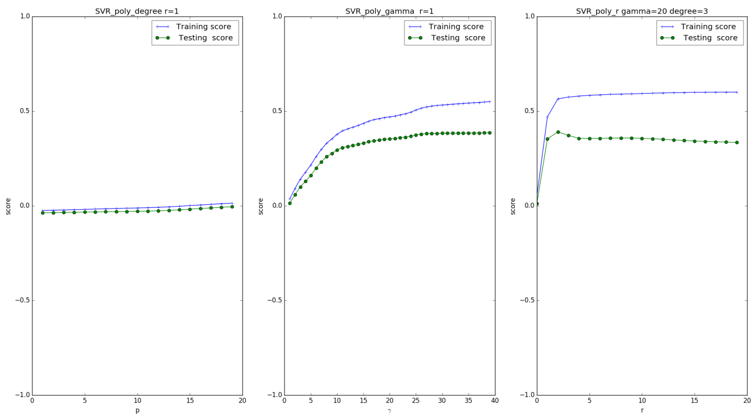


图 7.9 SVR_poly

接下来考察高斯核：$K(\vec{x},\vec{z})=\exp(-\gamma||\vec{x}-\vec{z}||^2)$，其中 γ 由 gamma参数决定。给出测试函数如下：

```
def test_SVR_rbf(*data):
    X_train,X_test,y_train,y_test=data
    gammas=range(1,20)
    train_scores=[]
    test_scores=[]
    for gamma in gammas:
        regr=svm.SVR(kernel='rbf',gamma=gamma)
        regr.fit(X_train,y_train)
```

```
        train_scores.append(regr.score(X_train,y_train))
        test_scores.append(regr.score(X_test, y_test))
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    ax.plot(gammas,train_scores,label="Training score ",marker='+' )
    ax.plot(gammas,test_scores,label= " Testing  score ",marker='o' )
    ax.set_title( "SVR_rbf")
    ax.set_xlabel(r"$\gamma$")
    ax.set_ylabel("score")
    ax.set_ylim(-1,1)
    ax.legend(loc="best",framealpha=0.5)
    plt.show()
```

测试结果如图 7.10 所示。可以看到在测试集上的预测性能随 γ 的变化比较平稳。

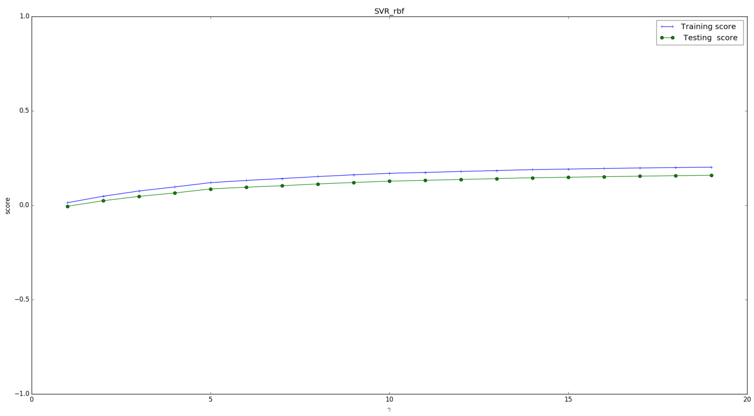


图 7.10 SVR_rbf

最后观察sigmoid核：$K(\vec{x},\vec{z})=\tanh(\gamma(\vec{x}\cdot\vec{z})+r)$。其中 γ 由 gamma参数决定，r 由 coef0参数指定。我们给出测试函数如下：

```
def test_SVR_sigmod(*data):
    X_train,X_test,y_train,y_test=data
    fig=plt.figure()

    ### 测试 gamma ####
    gammas=np.logspace(-1,3)
    train_scores=[]
    test_scores=[]

    for gamma in gammas:
        regr=svm.SVR(kernel='sigmoid',gamma=gamma,coef0=0.01)
        regr.fit(X_train,y_train)
        train_scores.append(regr.score(X_train,y_train))
        test_scores.append(regr.score(X_test, y_test))
```

```
ax=fig.add_subplot(1,2,1)
ax.plot(gammas,train_scores,label="Training score ",marker='+' )
ax.plot(gammas,test_scores,label= " Testing  score ",marker='o' )
ax.set_title( "SVR_sigmoid_gamma r=0.01")
ax.set_xscale("log")
ax.set_xlabel(r"$\gamma$")
ax.set_ylabel("score")
ax.set_ylim(-1,1)
ax.legend(loc="best",framealpha=0.5)
### 测试 r ######
rs=np.linspace(0,5)
train_scores=[]
test_scores=[]

for r in rs:
    regr=svm.SVR(kernel='sigmoid',coef0=r,gamma=10)
    regr.fit(X_train,y_train)
    train_scores.append(regr.score(X_train,y_train))
    test_scores.append(regr.score(X_test, y_test))
ax=fig.add_subplot(1,2,2)
ax.plot(rs,train_scores,label="Training score ",marker='+' )
ax.plot(rs,test_scores,label= " Testing  score ",marker='o' )
ax.set_title( "SVR_sigmoid_r gamma=10")
ax.set_xlabel(r"r")
ax.set_ylabel("score")
ax.set_ylim(-1,1)
ax.legend(loc="best",framealpha=0.5)
plt.show()
```

测试结果如图 7.11 所示。固定 $r = 0.01$ ，预测性能随着 γ 的增长而先增长后稳定；固定 $\gamma = 10$ ，预测性能则随着 r 的增长而下降。

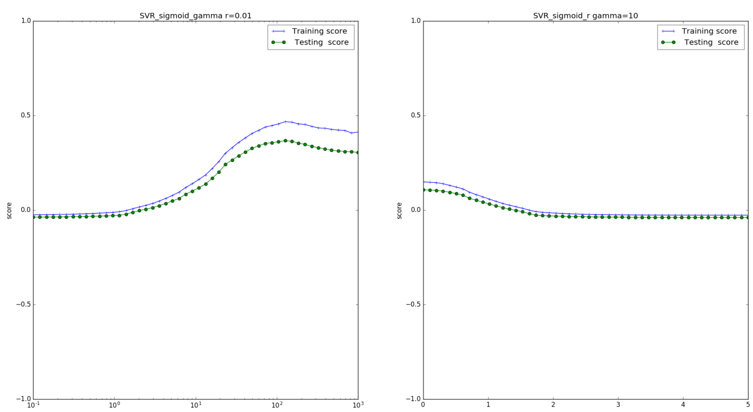


图 7.11　SVR_sigmoid

第 8 章

人工神经网络

8.1 概述

受生物学的启发，人工神经网络是由一系列简单的单元相互紧密联系构成的，每个单元有一定数量的实数输入和唯一的实数输出。神经网络的一个重要的用途就是接受和处理传感器产生的复杂的输入并进行自适应性的学习。人工神经网络算法模拟生物神经网络，是一种模式匹配算法，通常用于解决分类和回归问题。

人工神经网络是机器学习的一个庞大的分支，有几百种不同的算法。常见的人工神经网络算法包括：感知机神经网络 (Perceptron Neural Network)、反向传递 (Back Propagation, BP) 网络、Hopfield 网络、自组织映射 (Self-Organizing Map, SOM) 网络、学习矢量量化 (Learning Vector Quantization, LVQ) 网络等。

8.2 算法笔记精华

8.2.1 感知机模型

感知机是一种线性分类器，它用于二类分类问题。它将每一个实例分类为正类（取值 +1）和负类（取值 −1）。感知机的物理意义是：它将输入空间（特征空间）划分为正负两类的分离超平面。

感知机定义

设特征空间为 $\mathcal{X} \subseteq \mathbb{R}^n$，输出空间为 $\mathcal{Y} = \{+1, -1\}$。输入 $\vec{\boldsymbol{x}} \in \mathcal{X}$ 为特征空间的点；输出 $y \in \mathcal{Y}$ 为实例的类别。

定义函数：$f(\vec{\boldsymbol{x}}) = \text{sign}(\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} + b)$ 为感知机。其中 $\vec{\boldsymbol{w}} \in \mathbb{R}^n$ 为权值列向量，$b \in \mathbb{R}$ 为偏置，$\cdot$ 为向量内积。$\vec{\boldsymbol{w}}, b$ 为感知机的模型参数。$\text{sign}(x)$ 为示性函数，其定义如下：

$$\text{sign}(x) = \begin{cases} +1, & x \geqslant 0 \\ -1, & x < 0 \end{cases}$$

由于方程 $\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} + b = 0$ 给出了一个超平面，因此感知机对应特征空间 $\mathbb{R}^n$ 上的一个超平面 S。$\vec{\boldsymbol{w}}$ 是超平面 S 的法向量，b 是超平面的截距。超平面 S 将特征空间划分为两部分，因此超平面 S 也称为分离超平面：

- ❑ 超平面 S 上方的点判别为正类。
- ❑ 超平面 S 下方的点判别为负类。

感知机学习策略

给定数据集

$$T = \{(\vec{\boldsymbol{x}}_{\boldsymbol{1}}, y_1), (\vec{\boldsymbol{x}}_{\boldsymbol{2}}, y_2), \cdots, (\vec{\boldsymbol{x}}_N, y_N)\}, \vec{\boldsymbol{x}}_{\boldsymbol{i}} \in \mathcal{X} \subseteq \mathbb{R}^n, y_i \in \mathcal{Y} = \{+1, -1\}, i = 1, 2, \cdots, N$$

若存在某个超平面 $S: \vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} + b = 0$ 使得对数据集中的每一个实例 $(\vec{\boldsymbol{x}}_i, y_i)$ 有 $(\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}}_i + b) y_i > 0$，则称数据集 T 为线性可分数据集。线性可分意味着，将数据集中的正实例点与负实例点完全正确地划分到超平面的两侧。

定义感知机的损失函数为：误分类点到超平面 S 的总距离。虽然也可以将损失函数定义成误分类点的总数，但这种定义不是 $\vec{\boldsymbol{w}}, b$ 的连续可导函数，不容易优化，所以采用总距离。

- ❑ 对正确分类的样本点 $(\vec{\boldsymbol{x}}_i, y_i)$，有 $(\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}}_i + b) y_i > 0$。
- ❑ 对误分类的样本点 $(\vec{\boldsymbol{x}}_i, y_i)$，有 $(\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}}_i + b) y_i < 0$。

任取一个误分类的样本点 $(\vec{\boldsymbol{x}}_i, y_i)$，则 $\vec{\boldsymbol{x}}_i$ 距离超平面的距离为：

$$\frac{1}{||\vec{\boldsymbol{w}}||_2} |(\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}}_i + b)|$$

其中 $||\vec{\boldsymbol{w}}||_2$ 为 $\vec{\boldsymbol{w}}$ 的 L_2 范数。

考虑到 $|y_i| = 1$，以及对误分类点 $(\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}}_i + b) y_i < 0$，因此上式等于

$$\frac{-y_i(\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}}_i + b)}{||\vec{\boldsymbol{w}}||_2}$$

若不考虑 $\frac{1}{||\vec{w}||_2}$ (因为感知机要求训练数据集线性可分，最终误分类点数量为零，此时损失函数为零。即使考虑分母，也是零。若训练数据集线性不可分，则感知机算法无法收敛)，则得到感知机学习的损失函数的最终形式：

$$L(\vec{\boldsymbol{w}}, b) = -\sum_{\vec{\boldsymbol{x}}_i \in M} y_i(\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}}_i + b)$$

其中，M 为误分类点的集合。它隐式地与 $\vec{\boldsymbol{w}}, b$ 相关，因为 $\vec{\boldsymbol{w}}, b$ 优化导致误分类点减少从而使得 M 收缩。感知机的学习策略就是损失函数最小化。

从损失函数的定义可以得出损失函数的性质如下。

- 如果没有误分类点，则损失函数值为 0，因为 $M = \phi$。
- 误分类点越少或者误分类点距离超平面 S 越近，则损失函数 L 越小。
- 对于特定的样本点，其损失如下。
  - 若正确分类，则损失为 0。
  - 若误分类，则损失为 $\vec{\boldsymbol{w}}, b$ 的线性函数。
- 给定训练数据集 T ，损失函数 $L(\vec{\boldsymbol{w}}, b)$ 是 $\vec{\boldsymbol{w}}, b$ 的连续可导函数。

8.2.2　感知机学习算法

原始形式

给定训练数据集 $T = \{(\vec{\boldsymbol{x}}_{\boldsymbol{1}}, y_1), (\vec{\boldsymbol{x}}_{\boldsymbol{2}}, y_2), \cdots, (\vec{\boldsymbol{x}}_N, y_N)\}, \vec{\boldsymbol{x}}_{\boldsymbol{i}} \in \mathcal{X} \subseteq \mathbb{R}^n, y_i \in \mathcal{Y} = \{+1, -1\}, i = 1, 2, \cdots, N$，求参数 $\vec{\boldsymbol{w}}, b$，使得：

$$\min_{\vec{\boldsymbol{w}}, b} L(\vec{\boldsymbol{w}}, b) = \min_{\vec{\boldsymbol{w}}, b}\left[-\sum_{\vec{\boldsymbol{x}}_i \in M} y_i(\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}}_i + b)\right]$$

首先假设误分类点集合 M 是固定的，定义损失函数 $L(\vec{\boldsymbol{w}}, b)$ 的梯度：

$$\nabla_{\vec{\boldsymbol{w}}} L(\vec{\boldsymbol{w}}, b) = -\sum_{\vec{\boldsymbol{x}}_i \in M} y_i \vec{\boldsymbol{x}}_i \nabla_b L(\vec{\boldsymbol{w}}, b) = -\sum_{\vec{\boldsymbol{x}}_i \in M} y_i$$

利用梯度下降法，随机选取一个误分类点 $(\vec{\boldsymbol{x}}_i, y_i)$，对 $\vec{\boldsymbol{w}}, b$ 进行更新：

$$\vec{\boldsymbol{w}} \leftarrow \vec{\boldsymbol{w}} + \eta y_i \vec{\boldsymbol{x}}_i b \leftarrow b + \eta y_i$$

其中 $\eta \in (0, 1]$ 是步长，即学习率。通过迭代可以使得损失函数 $L(\vec{\boldsymbol{w}}, b)$ 不断减小直到为 0。每次修正一轮梯度，这就是梯度下降法的名字由来。步长 η 比较小的原因是：原本误分

类点的距离为

$$\frac{-y_i(\vec{\boldsymbol{w}}\cdot\vec{\boldsymbol{x}}_i+b)}{||\vec{\boldsymbol{w}}||_2}$$

如果步长较大，导致 $||\vec{\boldsymbol{w}}||$ 变化较大，那么再也无法忽略分母的影响。

这里得到下列感知机学习算法的原始形式。

- 输入：
  - 线性可分训练数据集 $T=\{(\vec{\boldsymbol{x}}_1,y_1),(\vec{\boldsymbol{x}}_2,y_2),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\}$；
  - 学习率 $\eta\in(0,1]$。
- 输出：感知机参数 $\vec{\boldsymbol{w}},b$。
- 算法步骤如下。
  - 选取初始值 $\vec{\boldsymbol{w}}_0,b_0$。
  - 在训练数据集中选取数据 $(\vec{\boldsymbol{x}}_i,y_i)$。
  - 若 $y_i(\vec{\boldsymbol{w}}\cdot\vec{\boldsymbol{x}}_i+b)\leqslant 0$（即该实例为误分类点）则更新参数：

$$\vec{\boldsymbol{w}}\leftarrow\vec{\boldsymbol{w}}+\eta y_i\vec{\boldsymbol{x}}_i b\leftarrow b+\eta y_i$$

  - 在训练数据集中重复选取数据来更新 $\vec{\boldsymbol{w}},b$ 直到训练数据集中没有误分类点为止。

对于某个点 $(\vec{\boldsymbol{x}}_i,y_i)$ 若选取它时，发现它是一个误分类点，则进行梯度下降后，设该点距新的超平面 S' 距离为 d'；而在梯度下降之前，设该点距旧的超平面 S 距离为 d。则：

$$\begin{aligned}\Delta d=d'-d&=\frac{1}{||\vec{\boldsymbol{w}}'||_2}|\vec{\boldsymbol{w}}'\cdot\vec{\boldsymbol{x}}_i+b'|-\frac{1}{||\vec{\boldsymbol{w}}||_2}|\vec{\boldsymbol{w}}^T\cdot\vec{\boldsymbol{x}}_i+b|\\&=-\frac{y_i}{||\vec{\boldsymbol{w}}'||_2}\left(\vec{\boldsymbol{w}}'\cdot\vec{\boldsymbol{x}}_i+b'\right)+\frac{y_i}{||\vec{\boldsymbol{w}}||_2}\left(\vec{\boldsymbol{w}}^T\cdot\vec{\boldsymbol{x}}_i+b\right)\\&\simeq-\frac{y_i}{||\vec{\boldsymbol{w}}||_2}\left[(\vec{\boldsymbol{w}}'-\vec{\boldsymbol{w}})\cdot\vec{\boldsymbol{x}}_i+(b'-b)\right]\\&=-\frac{y_i}{||\vec{\boldsymbol{w}}||_2}\left[\eta y_i\vec{\boldsymbol{x}}_i\cdot\vec{\boldsymbol{x}}_i+\eta y_i\right]\\&=-\frac{\eta y_i^2}{||\vec{\boldsymbol{w}}||_2}(\vec{\boldsymbol{x}}_i\cdot\vec{\boldsymbol{x}}_i+1)<0\end{aligned}$$

因此有 $d'<d$ 。这里要求 $\vec{\boldsymbol{w}}'\simeq\vec{\boldsymbol{w}}$，因此步长 η 要相当小。所以，每经过一轮迭代，被迭代的误分类点距离新的分类超平面的距离越近。

由此得到算法的几何解释：当一个实例点被误分类时，调整 $\vec{\boldsymbol{w}},b$ 使得分离平面向该误分类点的一侧移动，以减少该误分类点与超平面间的距离，直至超平面对所有的实例都能够正确分类为止。

感知机学习算法的原始形式获得的解并不是唯一的。感知机学习算法由于采用不同的初值或者误分类点选取顺序的不同，最终解可以不同。因为如果训练数据集是线性可分的，

则必然存在无数个分离超平面可以将训练数据集分开。若存在某个分离超平面 S 可以分离训练数据集，则给 S 一个微小的转动便可以构造一个新的分离超平面。

感知机收敛性定理：设训练数据集 $T=\{(\vec{\boldsymbol{x}}_{\boldsymbol{1}},y_1),(\vec{\boldsymbol{x}}_{\boldsymbol{2}},y_2),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\},\vec{\boldsymbol{x}}_{\boldsymbol{i}}\in\mathcal{X}\subseteq\mathbb{R}^n,y_i\in\mathcal{Y}=\{+1,-1\},i=1,2,\cdots,N$，其中 T 是线性可分的。则：

- 存在满足 $||\hat{\vec{\boldsymbol{w}}}_{opt}||_2=1$ 的超平面 $\hat{\vec{\boldsymbol{w}}}_{opt}\cdot\hat{\vec{\boldsymbol{x}}}=\vec{\boldsymbol{w}}_{opt}\cdot\vec{\boldsymbol{x}}+b_{opt}=0$ 将 T 完全正确分开。
  且存在 $\gamma>0$，对所有的 $i=1,2,\cdots,N$，有 $y_i(\hat{\vec{\boldsymbol{w}}}_{opt}\cdot\hat{\vec{\boldsymbol{x}}})=y_i(\vec{\boldsymbol{w}}_{opt}\cdot\vec{\boldsymbol{x}}+b_{opt})\geqslant\gamma$，其中 $\hat{\vec{\boldsymbol{w}}}=(\vec{\boldsymbol{w}},b),\hat{\vec{\boldsymbol{x}}}=(\vec{\boldsymbol{x}},1),\hat{\vec{\boldsymbol{w}}}\in\mathbb{R}^{n+1},\hat{\vec{\boldsymbol{x}}}\in\mathbb{R}^{n+1},\vec{\boldsymbol{w}}\in\mathbb{R}^n,\vec{\boldsymbol{x}}\in\mathbb{R}^n$。
- 令 $R=\max_{1\leqslant i\leqslant N}||\hat{\vec{\boldsymbol{x}}}_i||_2$，则感知机学习算法的原始形式在 T 上的误分类次数 k 满足：$k\leqslant(\frac{R}{\gamma})^2$。

对偶形式

从原始迭代形式

$$\vec{\boldsymbol{w}}\leftarrow\vec{\boldsymbol{w}}+\eta y_i\vec{\boldsymbol{x}}_i$$

$$b\leftarrow b+\eta y_i$$

如果考虑取初始值 $\vec{\boldsymbol{w}}_0=\vec{\boldsymbol{0}},b_0=0$，则迭代公式可以改写为：

$$\vec{\boldsymbol{w}}=\sum_{i=1}^{N}\alpha_i y_i\vec{\boldsymbol{x}}_i$$

$$b=\sum_{i=1}^{N}\alpha_i y_i$$

从而得到感知机学习算法的对偶形式。

- 输入：
  - 线性可分数据集 $T=\{(\vec{\boldsymbol{x}}_{\boldsymbol{1}},y_1),(\vec{\boldsymbol{x}}_{\boldsymbol{2}},y_2),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\}$；
  - 学习率 $\eta\in(0,1]$。
- 输出：
  - 感知机参数 $\vec{\alpha},b$；
  - 感知机模型 $f(\vec{\boldsymbol{x}})=\text{sign}(\sum_{j=1}^{N}\alpha_j y_j\vec{\boldsymbol{x}}_{\boldsymbol{j}}\cdot\vec{\boldsymbol{x}}+b)$；其中 $\vec{\alpha}=(\alpha_1,\alpha_2,\cdots,\alpha_N)^T$。
- 算法步骤如下：
  - $\vec{\alpha}\leftarrow\vec{\boldsymbol{0}},b\leftarrow 0$；
  - 在训练数据集中选取数据 $(\vec{\boldsymbol{x}}_i,y_i)$；
  - 若 $y_i(\sum_{j=1}^{N}\alpha_j y_j\vec{\boldsymbol{x}}_{\boldsymbol{j}}\cdot\vec{\boldsymbol{x}}_i+b)\leqslant 0$（即 $\vec{\boldsymbol{x}}_i$ 为误分类点），则更新参数：

$$\alpha_i \leftarrow \alpha_i + \eta$$
$$b \leftarrow b + \eta y_i$$

- 在训练数据集中重复选取数据来更新 $\vec{\alpha}, b$，直到训练数据集中没有误分类点为止。

在这里训练数据集 T 仅仅以内积的形式出现（因为算法只需要内积信息）。因此我们可以预先计算矩阵 $G = \left[\vec{x}_i \cdot \vec{x}_j\right]_{N\times N}$，即 Gram 矩阵，该矩阵给出了 T 中的实例间的内积。

可以证明，对于线性可分数据集，感知机学习算法的对偶形式收敛，且存在多个解。

8.2.3 神经网络

从感知机到神经网络

神经网络中最基本的成分是神经元，神经元的模型neuron描述如下。

- 每个神经元与其他神经元相连。
- 当一个神经元“兴奋”时，它会向相连的神经元发送化学物质。这样会改变相邻神经元内部的电位。
- 如果某个神经元的电位超过了一个“阈值”，则该神经元会被激活。

这样的神经元模型就是“M-P神经元模型”。在这个模型中：

- 每个神经元接收到来自相邻神经元传递过来的输入信号；
- 这些输入信号通过带权重的连接进行传递；
- 神经元接收到的总输入值将与神经元的阈值进行比较，然后通过“激活函数”处理以产生神经元输出；
- 理论上的激活函数为阶跃函数

$$f(x) = \begin{cases} 1, & x \geqslant 0 \\ 0, & x < 0 \end{cases}$$

其模型如图 8.1 所示，其中：

- $x_i, i = 1, 2, \cdots, n$ 为来自第 i 个相邻神经元的输入；
- $w_i, i = 1, 2, \cdots, n$ 为第 i 个相邻神经元的连接权重；
- θ 为当前神经元的阈值；
- $y = f(\sum_{i=1}^{n} w_i x_i - \theta)$ 为当前神经元的输出，f 为激活函数。

感知机可以看作神经网络的特例。感知机由两层神经元组成：输入层接收外界输入信号，输出层是M-P神经元。给定训练数据集 $T = \{(\vec{\boldsymbol{x}}_1, y_1), (\vec{\boldsymbol{x}}_2, y_2), \cdots, (\vec{\boldsymbol{x}}_N, y_N)\}, \vec{\boldsymbol{x}}_i \in \mathcal{X} \subseteq \mathbb{R}^n, y_i \in \mathcal{Y} = \{+1, -1\}, i = 1, 2, \cdots, N$。

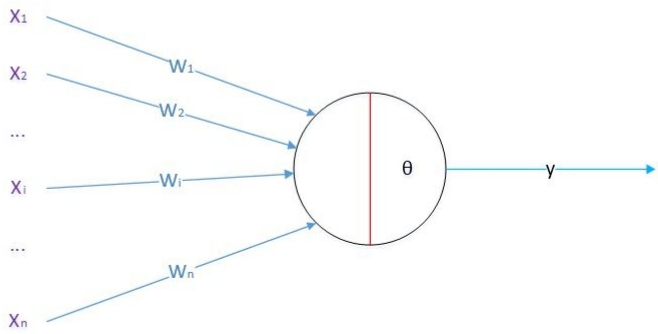


图 8.1 MP_neuron

设 $\vec{\boldsymbol{x}}_{\boldsymbol{i}} = (x_i^{(1)}, x_i^{(2)}, \cdots, x_i^{(n)})^T$，即 $x_i^{(j)}, i = 1, 2, \cdots, N; j = 1, 2, \cdots, n$ 表示第 i 个输入（它是一个向量）的第 j 个分量（它是一个标量）。

我们将阈值 θ 视为一个固定输入为 1.0 的“哑节点”，对应的连接权重为 b，新的阈值为 0，则对任意输入 $\vec{\boldsymbol{x}}_{\boldsymbol{i}}$，感知机网络如图 8.2 所示。

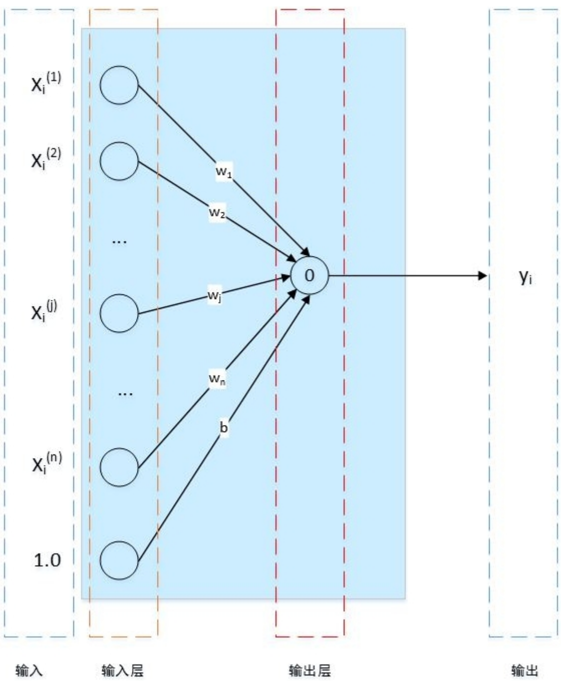


图 8.2 MP_perceptron

其输出为：$\hat{y}_i = f((\sum_{i=1}^{n} x_i^{(n)} w_i) + b) = f(\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}}_i + b)$。

注意：

- 感知机只有输出层神经元进行激活函数处理，即只拥有一层功能神经元。而输入层神经元并不进行激活函数处理。

- 感知机的激活函数为

$$f(x)=\begin{cases}1, & x\geqslant 0\\ -1, & x<0\end{cases}$$

多层前馈神经网络

感知机只拥有一层功能神经元，它只能处理线性可分问题。如果想解决非线性可分问题，则可以使用多层功能神经元，通常神经网络的结构如图 8.3 所示。

- 每层神经元与下一层神经元全部互连。
- 同层神经元之间不存在连接。
- 跨层神经元之间也不存在连接。

这样的神经网络结构通常称为“多层前馈神经网络”，其中输出层与输入层之间的一层神经元被称为隐层或者隐含层。多层前馈神经网络有以下特点：

- 隐含层和输出层神经元都是拥有激活函数的功能神经元；
- 输入层接收外界输入信号，不进行激活函数处理；
- 最终结果由输出层神经元给出。

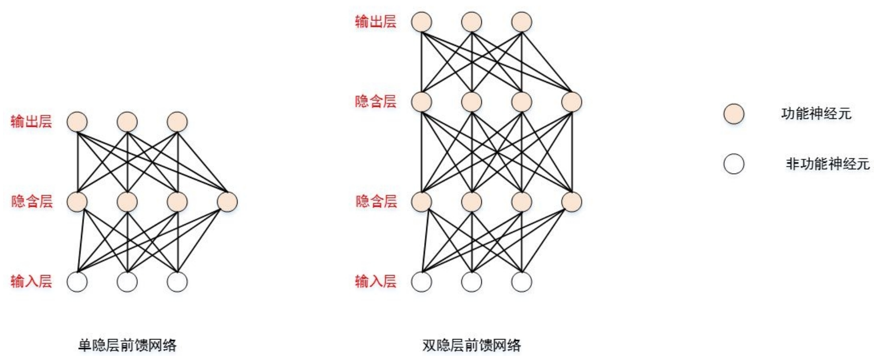


图 8.3 multi_layer_neural

神经网络的学习就是根据训练数据集来调整神经元之间的连接权重，以及每个功能神经元的阈值。

多层前馈神经网络的学习通常采用误差逆传播算法（error BackPropgation，BP）：该算法是训练多层神经网络的经典算法。

给定训练数据集 $T=\{(\vec{\boldsymbol{x}}_1,\vec{\boldsymbol{y}}_1),(\vec{\boldsymbol{x}}_2,\vec{\boldsymbol{y}}_2),\cdots,(\vec{\boldsymbol{x}}_N,\vec{\boldsymbol{y}}_N)\},\vec{\boldsymbol{x}}_i\in\mathcal{X}\subseteq\mathbb{R}^n,\vec{\boldsymbol{y}}_i\in\mathcal{Y}\subseteq\mathbb{R}^m,i=1,2,\cdots,N$，输入神经元为 n 个，输出神经元为 m 个。假设隐含层有 q 个神经元。设：

- 输出层第 j 个神经元的阈值用 θ_j 表示；
- 隐含层第 h 个神经元的阈值用 γ_h 表示；

- 输入层第 i 个神经元与隐含层第 h 个神经元的连接权重为 v_{ih}；
- 隐含层第 h 个神经元与输出层第 j 个神经元与的连接权重为 w_{hj}；
- 隐含层第 h 个神经元接收到的输入为 $\alpha_h = \sum_{i=1}^{n} v_{ih}x^{(i)}$，其中 $x^{(i)}$ 为输入特征向量 $\vec{x}$ 的第 i 个分量；
- 输出层第 j 个神经元接收到的输入为 $\beta_j = \sum_{h=1}^{q} w_{hj}b_h$，其中 b_h 为隐含层第 h 个神经元的输出，如图 8.4 所示。

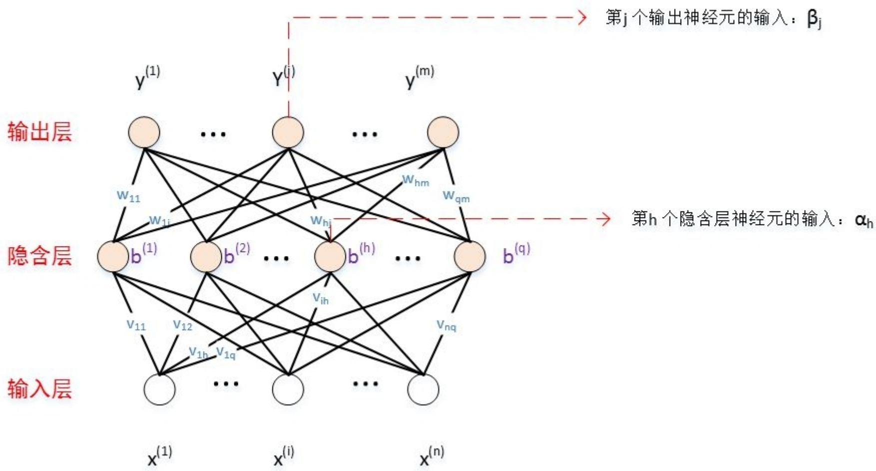


图 8.4 BP_neural

- Sigmoid函数（如图 8.5 所示）：

$$f(x) = \frac{1}{1+e^{-x}}$$

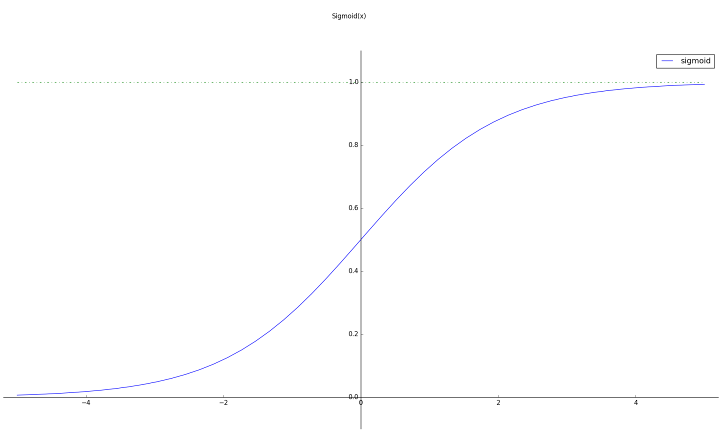


图 8.5 sigmoid

❑ 对训练样本 $(\vec{\boldsymbol{x}}_{\boldsymbol{k}}, \vec{\boldsymbol{y}}_{\boldsymbol{k}})$，假设神经网络的输出为 $\vec{\hat{\boldsymbol{y}}}_{\boldsymbol{k}} = (\hat{y}_k^{(1)}, \hat{y}_k^{(2)}, \cdots, \hat{y}_k^{(m)})^T$，即 $\hat{y}_k^{(j)} = f(\beta_j - \theta_j)$ 为输出向量 $\vec{\hat{\boldsymbol{y}}}_{\boldsymbol{k}}$ 的第 j 个分量，则网络在训练样本 $(\vec{\boldsymbol{x}}_{\boldsymbol{k}}, \vec{\boldsymbol{y}}_{\boldsymbol{k}})$ 上的均方误差为：

$$E_k = \frac{1}{2}\sum_{j=1}^{m}(\hat{y}_k^{(j)} - y_k^{(j)})^2$$

网络中需要确定下列参数：

❑ 输入层到隐含层的 $n \times q$ 个权值 v_{ih}，　$i = 1, 2, \cdots, n; h = 1, 2, \cdots, q$；
❑ 隐含层到输出层的 $q \times m$ 个权值 w_{hj}，　$h = 1, 2, \cdots, q; j = 1, 2, \cdots, m$；
❑ q 个隐含层神经元的阈值 γ_h，　$h = 1, 2, \cdots, q$；
❑ m 个输出层神经元的阈值 θ_j，　$j = 1, 2, \cdots, m$。

BP算法就是求解这 $q \times (n + m + 1) + m$ 个参数的算法。它的目标是最小化 E_k，具体方法是基于梯度下降算法，以目标的负梯度方向对参数进行调整：

$$\Delta w_{hj} = -\eta\frac{\partial E_k}{\partial w_{hj}}$$

$$\Delta \theta_j = -\eta\frac{\partial E_k}{\partial \theta_j}$$

$$\Delta v_{hj} = -\eta\frac{\partial E_k}{\partial v_{hj}}$$

$$\Delta \gamma_h = -\eta\frac{\partial E_k}{\partial \gamma_h}$$

其中 η 为学习率。

对上面这四个式子进行推导如下，以第一个式子为例：

考虑到 w_{hj} 首先影响的是第 j 个输出层神经元的输入值 β_j，然后再影响其输出值 $\hat{y}_k^{(j)}$，最后才影响到 E_k，因此有（数学依据是导数的链式法则）：

$$\frac{\partial E_k}{\partial w_{hj}} = \frac{\partial E_k}{\partial \hat{y}_k^{(j)}} \cdot \frac{\partial \hat{y}_k^{(j)}}{\partial \beta_j} \cdot \frac{\partial \beta_j}{\partial w_{hj}}$$

根据 $\beta_j = \sum_{h=1}^{q} w_{hj} b_h$ 这个定义式，有：

$$\frac{\partial \beta_j}{\partial w_{hj}} = b_h$$

根据Sigmoid函数的性质 $f'(x)=f(x)(1-f(x))$，以及定义式 $\hat{y}_k^{(j)}=f(\beta_j-\theta_j)$，有：

$$
\begin{aligned}
\frac{\partial E_k}{\partial \hat{y}_k^{(j)}}\cdot\frac{\partial \hat{y}_k^{(j)}}{\partial \beta_j} &= (\hat{y}_k^{(j)}-y_k^{(j)})\frac{\partial f(\beta_j-\theta_j)}{\partial \beta_j}\\
&= (\hat{y}_k^{(j)}-y_k^{(j)})f'(\beta_j-\theta_j)\\
&= (\hat{y}_k^{(j)}-y_k^{(j)})\hat{y}_k^{(j)}(1-\hat{y}_k^{(j)})
\end{aligned}
$$

令 $g_j=-\frac{\partial E_k}{\partial \beta_j}$ 为 E_k 对第 j 个输出神经元输入 β_j 的偏导数的相反数，是输出层神经元的梯度项。则：

$$
g_j=-\frac{\partial E_k}{\partial \hat{y}_k^{(j)}}\cdot\frac{\partial \hat{y}_k^{(j)}}{\partial \beta_j}=-\hat{y}_k^{(j)}(1-\hat{y}_k^{(j)})(\hat{y}_k^{(j)}-y_k^{(j)})
$$

于是得到BP算法中关于 w_{hj} 的更新公式：$\Delta w_{hj}=\eta g_j b_h$

类似可以得到

$$
\begin{aligned}
&\Delta\theta_j=-\eta g_j\\
&\Delta v_{ih}=\eta e_h x^{(i)}\\
&\Delta\gamma_h=-\eta e_h
\end{aligned}
$$

上式中的 $e_h=-\frac{\partial E_k}{\partial \alpha_h}$ 为 E_k 对第 h 个隐含层神经元的输入 α_h 的偏导数的相反数，为隐含层神经元的梯度项。简化 e_h 的前提如下。

- 根据 b_h 的定义 $b_h=f(\alpha_h-\gamma_h)$ 以及 Sigmoid 函数 f 的性质。
- 根据

$$
\frac{\partial E_k}{\partial b_h}=\sum_{j=1}^{m}\frac{\partial E_k}{\partial \beta_j}\frac{\partial \beta_j}{\partial b_h}
$$

  其意义为：第 h 个隐含层神经元的输出 b_h 会传导至所有的输出层神经元的输入 $\beta_1,\beta_2,\cdots,\beta_m$；而输出层神经元的输入 β_j 又会传导至均方误差 E_k 。因此 E_k 对 b_h 的偏导数需要考虑输出层所有的神经元输入 $\beta_j, j=1,2,\cdots,m$。
- 根据定义式

$$
g_j=-\frac{\partial E_k}{\partial \beta_j}
$$

- 根据定义式

$$
\beta_j=\sum_{h=1}^{q}w_{hj}b_h\rightarrow\frac{\partial \beta_j}{\partial b_h}=w_{hj}
$$

则有：

$$
\begin{aligned}
e_h &= -\frac{\partial E_k}{\partial \alpha_h} = -\frac{\partial E_k}{\partial b_h} \cdot \frac{\partial b_h}{\partial \alpha_h} \\
&= -\sum_{j=1}^{m} \frac{\partial E_k}{\partial \beta_j} \frac{\partial \beta_j}{\partial b_h} f'(\alpha_h - \gamma_h) = \sum_{j=1}^{m} g_j w_{hj} b_h (1 - b_h) \\
&= b_h (1 - b_h) \sum_{j=1}^{m} g_j w_{hj}
\end{aligned}
$$

> BP算法从原理上就是普通的梯度下降法求最小值的问题。它关键的地方在以下两个：
>
> - 导数的链式法则；
> - sigmoid激活函数的性质：sigmoid函数求导的结果等于自变量的乘积形式。

根据BP算法的原理，得到误差逆传播算法。

- 输入：
  - 训练数据集 $T = \{(\vec{\boldsymbol{x}}_1, \vec{\boldsymbol{y}}_1), (\vec{\boldsymbol{x}}_2, \vec{\boldsymbol{y}}_2), \cdots, (\vec{\boldsymbol{x}}_N, \vec{\boldsymbol{y}}_N)\}, \vec{\boldsymbol{x}}_i \in \mathcal{X} \subseteq \mathbb{R}^n, \vec{\boldsymbol{y}}_i \in \mathcal{Y} \subseteq \mathbb{R}^m, i = 1, 2, \cdots, N$；
  - 学习率 η。
- 输出：
  - 输入层到隐含层的 $n \times q$ 个权值 $v_{ih}, i = 1, 2, \cdots, n; h = 1, 2, \cdots, q$；
  - 隐含层到输出层的 $q \times m$ 个权值 $w_{hj}, h = 1, 2, \cdots, q; j = 1, 2, \cdots, m$；
  - q 个隐含层神经元的阈值 $\gamma_h, h = 1, 2, \cdots, q$；
  - m 个输出层神经元的阈值 $\theta_j, j = 1, 2, \cdots, m$。
- 算法步骤如下。
  - 在 0, 1 范围内随机初始化网络中所有的连接权值和阈值。
  - 对训练数据集 T 中的样本点迭代，直到到达停止条件为止，迭代过程如下：
    - 取样本点 $(\vec{\boldsymbol{x}}_k, \vec{\boldsymbol{y}}_k)$，根据当前参数计算该样本的输出 $\hat{\vec{\boldsymbol{y}}}_k = (\hat{y}_k^{(1)}, \hat{y}_k^{(2)}, \cdots, \hat{y}_k^{(m)})^T$。

      > 这一步是正向传播过程：输入样本 → 输入层 → 隐含层 → 输出层。

    - 根据 $g_j = -\hat{y}_k^{(j)}(1 - \hat{y}_k^{(j)})(\hat{y}_k^{(j)} - y_k^{(j)})$，计算输出层神经元的梯度项 g_j，其中 $\vec{\boldsymbol{y}}_k = (y_k^{(1)}, y_k^{(2)}, \cdots, y_k^{(m)})^T$。

      > 在这一步计算出了输出层神经元的误差。

    - 根据 $e_h = b_h(1 - b_h)\sum_{j=1}^{m} g_j w_{hj}$ 计算隐层神经元的梯度项 e_h。

> 这一步将误差逆向传播到了隐含层神经元，这就是误差逆传播的来历。

❖ 根据下列式子更新连接权值 w_{hj}, v_{ih} 和阈值 θ_j, γ_h：

> 这一步根据隐含层神经元的误差来对连接权值和阈值进行调整。通过将输出误差逆向传播，将误差分摊给各层所有单元，从而获得各层单元的误差信号，进而修正各单元的权值（其过程是一个权值调整的过程）。

$$\Delta w_{hj} = \eta g_j b_h$$
$$\Delta \theta_j = -\eta g_j$$
$$\Delta v_{ih} = \eta e_h x^{(i)}$$
$$\Delta \gamma_h = -\eta e_h$$

> 通常的停止条件是：训练样本集上的累计误差 $E = \frac{1}{N}\sum_{k=1}^{N} E_k$ 达到一个很小的值。

上面介绍的误差逆传播算法是“标准BP算法”，它是基于单个的 $E_k = \frac{1}{2}\sum_{j=1}^{m}(\hat{y}_k^{(j)} - y_k^{(j)})^2$ 最小化原则来推导的。还有一种算法称为“累计误差逆传播算法”。累计误差逆传播算法最小化的目标是：训练数据集 T 上的累计误差：$E = \frac{1}{N}\sum_{k=1}^{N} E_k$。其推导过程类似于标准 BP 算法。标准 BP 算法和累计BP算法都经常使用。两者的区别如下：

- 标准BP算法需要进行更多次数的迭代。因为标准BP算法每次更新只针对单个样例，参数更新非常频繁，而且对于不同的样例进行更新的效果可能出现“抵消”现象；
- 累计BP算法参数更新频率低得多。因为累计BP算法直接针对累计误差最小化，它在读取整个一轮训练集 T 之后才对参数进行一次更新。

> 但是在很多任务中，训练集的累计误差下降到一定程度之后，进一步的下降会非常缓慢。这时标准BP往往会更快获得较好的解。

可以证明：多层前馈网络若包含足够多神经元的隐含层，则它就能够以任意精度逼近任意复杂度的连续函数。

事实上如何设置隐含层神经元的个数是个待解决的问题。实际应用中通常靠“试错法”调整参数，但缺乏理论支持。

BP神经网络表达能力非常强大，因此可能遇到过拟合：其训练误差降低，但是测试误差上升。这时有以下两种策略来缓解过拟合。

- “早停”策略：将数据集分成训练数据集和验证数据集两类。训练数据集用于计算梯度、更新连接权值和阈值；验证集用于估计误差。如果训练集误差降低但是验证集误差升高，则停止训练。同时返回具有最小验证集误差的连接权值和阈值。
- “正则化”策略：修改误差目标函数为：

$$E = \frac{1}{N}\sum_{k=1}^{N} E_k + \lambda \sum_i w_i^2$$

其中 w_i 表示连接权值和阈值；$\lambda > 0$ 表示对经验误差和网络复杂度的折中。$\lambda \sum_i w_i^2$ 则刻画了网络复杂度。

8.3 Python 实战

这里需要导入的包为：

```
from matplotlib import  pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import load_iris
```

8.3.1 感知机学习算法的原始形式

给出生成线性可分数据集的生成算法：

```
def creat_data(n):
    np.random.seed(1)
    x_11=np.random.randint(0,100,(n,1))
    x_12=np.random.randint(0,100,(n,1,))
    x_13=20+np.random.randint(0,10,(n,1,))
    x_21=np.random.randint(0,100,(n,1))
    x_22=np.random.randint(0,100,(n,1))
    x_23=10-np.random.randint(0,10,(n,1,))

    new_x_12=x_12*np.sqrt(2)/2-x_13*np.sqrt(2)/2## 沿X轴旋转45°
    new_x_13=x_12*np.sqrt(2)/2+x_13*np.sqrt(2)/2## 沿X轴旋转45°
    new_x_22=x_22*np.sqrt(2)/2-x_23*np.sqrt(2)/2## 沿X轴旋转45°
    new_x_23=x_22*np.sqrt(2)/2+x_23*np.sqrt(2)/2## 沿X轴旋转45°

    plus_samples=np.hstack([x_11,new_x_12,new_x_13,np.ones((n,1))])
    minus_samples=np.hstack([x_21,new_x_22,new_x_23,-np.ones((n,1))])
    samples=np.vstack([plus_samples,minus_samples])
    np.random.shuffle(samples)  # 混洗数据
    return samples
```

- 参数
  - n：正类的样本点数量，也是负类的样本点数量。总的样本点数量为 $2n$。
- 返回值：所有的样本点组成的数组，形状为 (2*n,4)。数组中的每一行代表一个样本点，由其特征 $\vec{x}$ 和标记 y 组成。

其过程为：首先在z轴坐标为 20 的上方生成 n 个随机点作为正类，在z轴坐标为 10 的下方生成 n 个随机点作为负类。此时在平面 $z=10, z=20$ 作为隔离带。然后 45° 旋转 x 坐标轴，再返回这些点在新坐标轴中的坐标。注意这里混洗了数据，否则会发现数据集的前半部分都是正类，后半部分都是负类，需要混洗数据从而让正负类交叉出现。

绘制数据集的函数为：

```
def plot_samples(ax,samples):
    Y=samples[:,-1]
    Y=samples[:,-1]
    position_p=Y==1 ## 正类位置
    position_m=Y==-1 ## 负类位置
    ax.scatter(samples[position_p,0],samples[position_p,1],
        samples[position_p,2],marker='+',label='+',color='b')
    ax.scatter(samples[position_m,0],samples[position_m,1],
        samples[position_m,2],marker='^',label='-',color='y')
```

- 参数
  - ax：一个 Axes3D实例，负责绘制图形。
  - samples：代表训练数据集的数组，形状为 (N,n_features+1)，其中N为样本点的个数，n_features代表特征数量（这里为 3，表示三个特征）。

plot_samples函数的用法为：

```
fig=plt.figure()
ax=Axes3D(fig)
data=creat_data(100)
plot_samples(ax,data)
ax.legend(loc='best')
plt.show()
```

然后给出感知机学习算法的原始形式算法的函数（图形如图 8.6 所示）：

```
def perceptron(train_data,eta,w_0,b_0):
     x=train_data[:,:-1]  # x 数据
     y=train_data[:,-1]  #  对应的分类
     length= train_data.shape[0] #样本集大小
     w=w_0
     b=b_0
     step_num=0
     while True:
```

```
        i=0
        while(i< length): ## 遍历一轮样本集中所有的样本点
            step_num+=1
            x_i=x[i].reshape((x.shape[1],1))
            y_i=y[i]
            if y_i*(np.dot(np.transpose(w),x_i)+b) <=0: # 该点是误分类点
                w=w+eta*y_i*x_i  # 梯度下降
                b=b+eta*y_i                 # 梯度下降
                break # 执行下一轮筛选
            else:#该点不是误分类点，选取下一个样本点
                i=i+1
        if(i== length): #没有误分类点，结束循环
            break
    return (w,b,step_num)
```

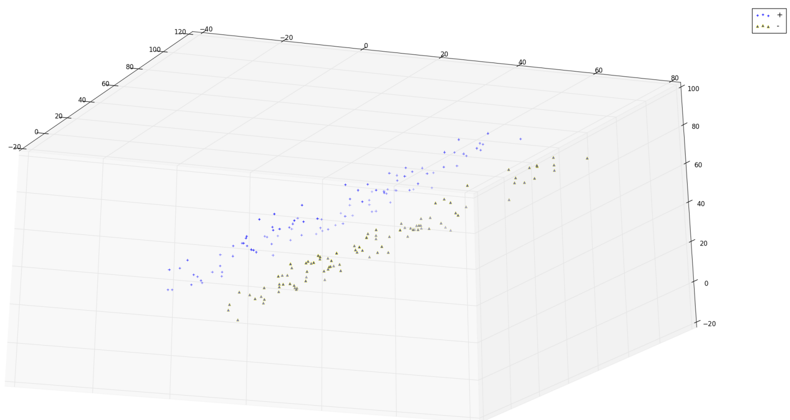


图 8.6 perceptron_data

- ❑ 参数
  - ❍ train_data：代表训练数据集的数组，形状为 (N,n_features+1)，其中N为样本点的个数，n_features代表特征数量（这里为 3，表示三个特征）。
  - ❍ eta：学习率。
  - ❍ w_0：即 $\vec{w}_0$，是一个列向量。
  - ❍ b_0：即 b_0，是一个标量。
- ❑ 返回值：一个元组，成员为 $\vec{w}, b$ 以及迭代次数。

其过程为：

- ❑ 最外层循环只有在全部分类正确的这种情况下退出；
- ❑ 内层循环从前到后遍历所有的样本点。一旦发现某个样本点是误分类点，就更新 $\vec{w}, b$，然后重新从头开始遍历所有的样本点。

由于需要绘制分离超平面，因此需要根据 $\vec{\boldsymbol{w}}, b$ 给出生成分离超平面的函数：

```
def creat_hyperplane(x,y,w,b):
        return (-w[0][0]*x-w[1][0]*y-b)/w[2][0]
```

- 参数
  - x：分离超平面上点的 x 坐标组成的数组。
  - y：分离超平面上点的 y 坐标组成的数组。
  - w：即 $\vec{\boldsymbol{w}}$ ，超平面的法向量，它是一个列向量。
  - b：即 b，超平面的截距。
- 返回值：分离超平面上点的 z 坐标组成的数组。

其过程就是根据 $w_x x + w_y y + w_z z + b = 0$ 这个方程求得的。

综合上述函数，可以观察感知机学习算法的原始算法的运行情况：

```
data=creat_data(100)
eta,w_0,b_0=0.1,np.ones((3,1),dtype=float),1
w,b,num=perceptron(data,eta,w_0,b_0)

fig=plt.figure()
plt.suptitle("perceptron")
ax=Axes3D(fig)

### 绘制样本点
plot_samples(ax,data)

##  绘制分离超平面
x=np.linspace(-30,100,100)  # 分离超平面的 x坐标数组
y=np.linspace(-30,100,100) # 分离超平面的 y坐标数组
x,y=np.meshgrid(x,y) # 划分网格
z=creat_hyperplane(x,y,w,b)  # 分离超平面的 z坐标数组
ax.plot_surface(x, y, z, rstride=1, cstride=1,color='g',alpha=0.2)

ax.legend(loc="best")
plt.show()
```

算法得到的 $\vec{\boldsymbol{w}}$ 为 [[-10.1] [-68.08433252][64.85174234]]，分离超平面法向量为 (-10.1, -68.08,64.85)，它在y-z平面上的投影是一条直线，该直线的斜率为68.08/64.85=1.05，非常接近我们在生成数据时旋转 45° 角的设定。感知机学习算法的原始形式算法的函数 perceptron_original 图形（如图 8.7 所示）。

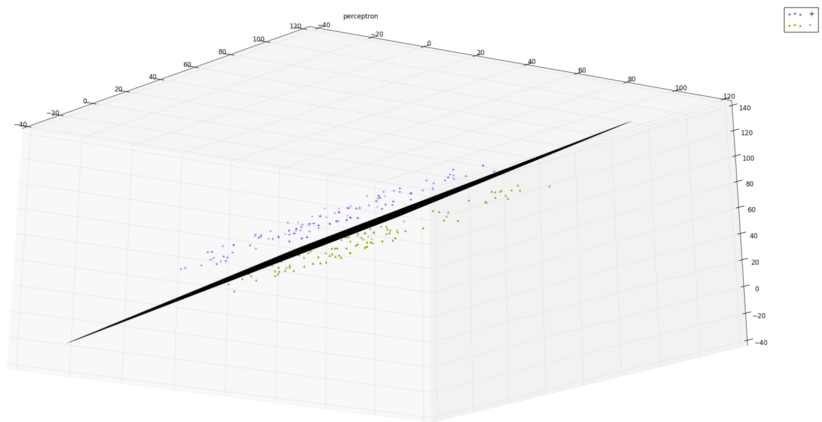


图 8.7 perceptron_original

8.3.2 感知机学习算法的对偶形式

给出感知机学习算法的对偶形式算法的函数：

```
def creat_w(train_data,alpha):
    x=train_data[:,:-1]  # x 数据
    y=train_data[:,-1]  #  对应的分类
    N= train_data.shape[0] #样本集大小
    w=np.zeros((x.shape[1],1))
    for i in range(0,N):
        w=w+alpha[i][0]*y[i]*(x[i].reshape(x[i].size,1))
    return w
def perceptron_dual(train_data,eta,alpha_0,b_0):
    x=train_data[:,:-1]  # x 数据
    y=train_data[:,-1]  #  对应的分类
    length= train_data.shape[0] #样本集大小
    alpha=alpha_0
    b=b_0
    step_num=0
    while True:
        i=0
        while(i< length):
            step_num+=1
            x_i=x[i].reshape((x.shape[1],1))
            y_i=y[i]
            w=creat_w(train_data,alpha)
            z=y_i*(np.dot(np.transpose(w),x_i)+b)
            if z <=0: # 该点是误分类点
                alpha[i][0]+=eta  # 梯度下降
                b+=eta*y_i  # 梯度下降
```

```
            break # 梯度下降了，从头开始，执行下一轮筛选
        else:
            i=i+1 #该点不是误分类点，选取下一个样本点
    if(i== length ): #没有误分类点，结束循环
        break
return (alpha,b,step_num)
```

这里有如下两个函数。

- creat_w：用于根据训练数据集和 $\vec{\alpha}$ 得到 $\vec{\boldsymbol{w}}$ ，这是因为在计算中大量地需要 $\vec{\boldsymbol{w}}$。
  - 参数
    - train_data：代表训练数据集的数组，形状为 (N,n_features+1)，其中N为样本点的个数，n_features代表特征数量（这里为 3，表示三个特征）。
    - alpha：是一个代表 $\vec{\alpha}$ 的数组，形状为 (N,1)，其中N为样本点的个数。
  - 返回值：代表 $\vec{\boldsymbol{w}}$ 的数组，形状为 (n_features,1)，n_features代表特征数量（这里为 3，表示三个特征）。
- perceptron_dual：用于感知机学习算法的对偶形式（如图 8.8 所示）。

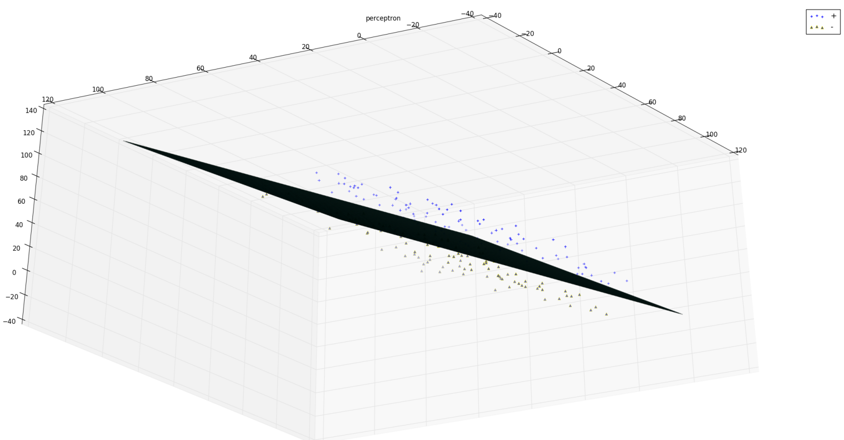


图 8.8　perceptron_dual

  - 参数
    - train_data：代表训练数据集的数组，形状为 (N,n_features+1)，其中N为样本点的个数，n_features代表特征数量（这里为 3，表示三个特征）。
    - eta：学习率。
    - alpha_0：即 $\vec{\alpha}_0$ ，是一个列向量。
    - b_0：即 b_0，是一个标量。
  - 返回值：一个元组，成员为 $\vec{\alpha}, b$ 以及迭代次数。

可以观察感知机学习算法的原始形式和对偶形式的运行情况如下：

```
data=creat_data(100)
eta,w_0,b_0=0.1,np.ones((3,1),dtype=float),1
w_1,b_1,num_1=perceptron(data,eta,w_0,b_0)
alpha,b_2,num_2=perceptron_dual(data,eta=0.1,alpha_0=np.zeros((data.shape[0]*2,1)),
    b_0=0)
w_2=creat_w(data,alpha)

print("w_1,b_1",w_1,b_1)
print("w_2,b_2",w_2,b_2)

fig=plt.figure()
plt.suptitle("perceptron")
ax=Axes3D(fig)

### 绘制样本点
plot_samples(ax,data)

##  绘制分离超平面
x=np.linspace(-30,100,100)  # 分离超平面的 x坐标数组
y=np.linspace(-30,100,100) # 分离超平面的 y坐标数组
x,y=np.meshgrid(x,y) # 划分网格
z=creat_hyperplane(x,y,w_1,b_1)  # 原始形式算法的分离超平面的 z坐标数组
z_2=creat_hyperplane(x,y,w_2,b_2)  # 对偶形式算法的分离超平面的 z坐标数组
ax.plot_surface(x, y, z, rstride=1, cstride=1,color='g',alpha=0.2)
ax.plot_surface(x, y, z_2, rstride=1, cstride=1,color='c',alpha=0.2)
ax.legend(loc="best")
plt.show()
```

原始形式算法得到的 $(\vec{\boldsymbol{w}},b)$ 为 ([-10.1,-68.08433252,64.85174234] -651.4)，对应的超平面为：$-0.156x-1.05y+z-10.04=0$；对偶形式算法得到的 $(\vec{\boldsymbol{w}},b)$ 为 ([-10.1,-67.17514421, 64.06387437] -641.9)，对应的超平面为：$-0.158x-1.05y+z-10.01=0$ 。可以看到这两者几乎完全重合。

另外观察结果的 $\vec{\alpha}$ ，发现绝大部分的分量为 0：

```
[[  7.00000000e-01] [  3.67000000e+01] [  5.00000000e-01] [  4.00000000e-01]
 [  5.00000000e-01] [  3.36100000e+02] [  1.34000000e+01] [  1.60000000e+00]
 [  1.25100000e+02] [  9.00000000e-01] [  0.00000000e+00] [  3.99700000e+02]
 [  0.00000000e+00] [  0.00000000e+00] [  0.00000000e+00] [  7.00000000e-01]
 [  0.00000000e+00] [  0.00000000e+00] [  4.00000000e-01] [  0.00000000e+00]
 [  0.00000000e+00] [  0.00000000e+00] [  0.00000000e+00] [  0.00000000e+00]
 [  0.00000000e+00] [  0.00000000e+00] [  0.00000000e+00] [  0.00000000e+00]
 [  0.00000000e+00] [  0.00000000e+00] [  7.20000000e+00] [  0.00000000e+00]
.....
 [  0.00000000e+00] [  0.00000000e+00] [  0.00000000e+00] [  0.00000000e+00]
```

```
[  0.00000000e+00] [  0.00000000e+00] [  0.00000000e+00] [  0.00000000e+00]
[  0.00000000e+00] [  0.00000000e+00] [  0.00000000e+00] [  0.00000000e+00]
[  0.00000000e+00] [  0.00000000e+00] [  0.00000000e+00] [  0.00000000e+00]]
```

这说明：大多数的样本点对于最终解并没有贡献。分离超平面的位置是由少部分重要的样本点决定的。而感知机学习算法的对偶形式能够找出这些重要的样本点。这就是支持向量机的原理。

8.3.3 学习率与收敛速度

下面来观察 η 参数对于模型学习收敛速度的影响。首先给出函数（如图 8.9 所示）：

```
def test_eta(data,ax,etas,w_0,alpha_0,b_0):
        nums1=[]
        nums2=[]
        for eta in etas:
            _,_,num_1=perceptron(data,eta,w_0=w_0,b_0=b_0)
            _,_,num_2=perceptron_dual(data,eta=0.1,alpha_0=alpha_0,b_0=b_0)
            nums1.append(num_1)
            nums2.append(num_2)
        ax.plot(etas,np.array(nums1),label='orignal iteraton times')
        ax.plot(etas,np.array(nums2),label='dual iteraton times')
```

参数

- data：训练数据集。
- ax：负责算法结果曲线绘制的Axes实例。
- etas：多个 η 值组成的列表。
- w_0：原始算法用到的 $\vec{\boldsymbol{w}}_0$。
- alpha_0：对偶算法用到的 $\vec{\alpha}_0$。
- b_0：原始算法和对偶算法均用到的 b_0。

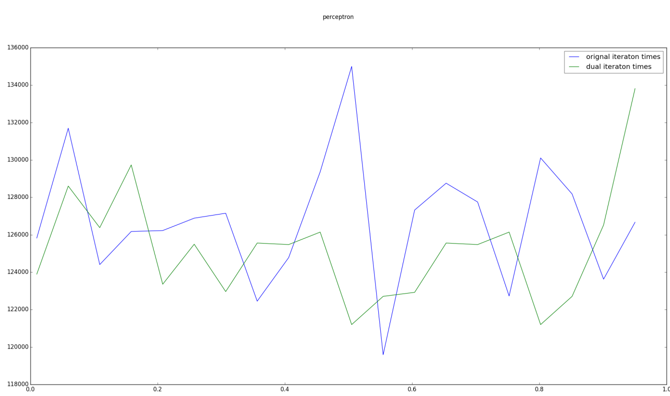


图 8.9 perceptron_eta

接下来开始使用该函数来查看 η 对收敛速度的影响：

```
fig=plt.figure()
fig.suptitle("perceptron")
ax=fig.add_subplot(1,1,1)
ax.set_xlabel(r'$\eta$')

data=creat_data(20)
etas=np.linspace(0.01,1,num=25,endpoint=False)
w_0,b_0,alpha_0=np.ones((3,1)),0,np.zeros((data.shape[0],1))
test_eta(data,ax,etas,w_0,alpha_0,b_0)

ax.legend(loc="best",framealpha=0.5)
plt.show()
```

注意：将训练数据集大小设定为 40（正类 20 个，负类 20 个），然后 η 为 0.01 ~ 1（一共 25 个）。如果取值太大会导致运算时间过长。可以发现：

- 感知机学习算法的原始形式和对偶形式的收敛速度近似相同，都在同一个数量级；
- 不同的 η 对于学习算法的收敛速度有影响，但是影响不大。

梯度下降算法每次迭代，都会受到学习速率 η 的影响，学习速率 η 大小的选择，总结如下：

1. 如果 η 较小，则达到收敛所需要迭代的次数就会非常高；
2. 如果 η 较大，则每次迭代可能不会减小代价函数的结果，甚至会超过局部最小值导致无法收敛。

通常的经验是，从以下几个数值开始试验 η 的值，0.001 ,0.003, 0.01, 0.03, 0.1, 0.3, 1, … η 初始值为 0.001, 不符合预期乘以 3 倍用 0.003 代替，不符合预期再用 0.01 替代，如此循环直至找到最合适的 η，然后对于这些不同的 η 值，绘制误差随迭代步数变化的曲线，然后选择看上去使得误差快速下降的一个 η 值。

8.3.4 感知机与线性不可分数据集

我们来观察感知机学习算法用于线性不可分数据集的情形。首先给出生成线性不可分数据集的函数：

```
def creat_data_no_linear(n):
    np.random.seed(1)
    x_11=np.random.randint(0,100,(n,1))
    x_12=np.random.randint(0,100,(n,1,))
    x_13=10+np.random.randint(0,10,(n,1,))
    x_21=np.random.randint(0,100,(n,1))
    x_22=np.random.randint(0,100,(n,1))
```

```
x_23=20-np.random.randint(0,10,(n,1,))

new_x_12=x_12*np.sqrt(2)/2-x_13*np.sqrt(2)/2## 沿X轴旋转45度
new_x_13=x_12*np.sqrt(2)/2+x_13*np.sqrt(2)/2## 沿X轴旋转45度
new_x_22=x_22*np.sqrt(2)/2-x_23*np.sqrt(2)/2## 沿X轴旋转45度
new_x_23=x_22*np.sqrt(2)/2+x_23*np.sqrt(2)/2## 沿X轴旋转45度

plus_samples=np.hstack([x_11,new_x_12,new_x_13,np.ones((n,1))])
minus_samples=np.hstack([x_21,new_x_22,new_x_23,-np.ones((n,1))])
samples=np.vstack([plus_samples,minus_samples])
np.random.shuffle(samples)  # 混洗数据
return samples
```

其参数和用法与creat_data相同。步骤也相同，唯一的区别是：随机数区间有所变化导致正类和负类的样本相互交叉。

绘制这个线性不可分数据集（如图 8.10 所示）：

```
data=creat_data_no_linear(100)
fig=plt.figure()
ax=Axes3D(fig)
plot_samples(ax,data)
ax.legend(loc='best')
plt.show()
```

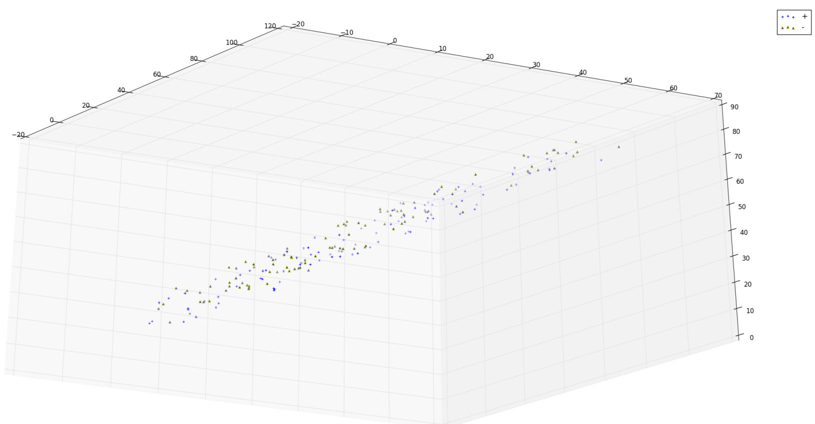


图 8.10　perceptron_data_no_linear

修改感知机学习算法原始形式的函数，将：

```
step_num+=1
```

这一行修改为：

```
step_num+=1
if step_num>=10000000:
```

```
print("failed!,step_num =%d"%step_num)
return
```

在这个线性不可分数据集上，运行感知机学习算法的原始形式：

```
data=creat_data_no_linear(100)
    perceptron(data,eta=0.1,w_0=np.zeros((2,1)),b_0=0)
```

得到输出：failed!,step_num =10000000。说明学习算法不收敛（迭代次数超过一千万次），因此感知机学习算法只能够用于线性可分数据集。

8.3.5 多层神经网络

对于多层神经网络，scikit-learn提供了MLPClassifier类。其初始化函数为：

```
    sklearn.neural_network.MLPClassifier(hidden_layer_sizes=(100, ),
activation='relu', algorithm='adam', alpha=0.0001, batch_size='auto',
learning_rate='constant', learning_rate_init=0.001, power_t=0.5, max_iter=200,
 shuffle=True, random_state=None, tol=0.0001, verbose=False,
 warm_start=False, momentum=0.9, nesterovs_momentum=True,
early_stopping=False, validation_fraction=0.1, beta_1=0.9, beta_2=0.999,
epsilon=1e-08)
```

MLPClassifier类实现了多层感知机算法 (multi-layer perceptron (MLP))，通过BP算法训练模型。

这里重点介绍几个重要的参数，剩下的参数请参考scikit-learn的官方文档。

- hidden_layer_sizes：是一个元组。元组指定了隐含层的结构。元组的长度表示隐含层的层数，元组的元素则指定了每一层隐含层中功能神经元的数量。
  - 如(97,98)表示隐含层有两层：第一层隐含层有 97 个功能神经元，第二层隐含层有 98 个功能神经元。
  - 默认参数的隐含层只有一层，该层有 100 个功能神经元。
- activation：是个字符串。该参数指定了激活函数的类型。可以为以下值。
  - 'logistic'：激活函数为sigmoid函数 $f(x)=\frac{1}{1+\exp(-x)}$。
  - 'tanh'：激活函数为tanh函数 $f(x)=\tanh(x)=\frac{\sinh(x)}{\cosh(x)}=\frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$。
  - 'relu'：激活函数为修正线性单元 $f(x)=\max(0,x)$。该值为默认参数值。
- algorithm：是个字符串。该参数指定了采用的最优化算法的类型。可以为下列值。
  - 'l-bfgs'：一种伪牛顿算法。对于小规模的数据集，这种算法效果较好。
  - 'sgd'：随机梯度下降法。
  - 'adam'：由Kingma, Diederik, and Jimmy Ba提出的一种stochastic gradient-based最优化算法。对于相对较大规模数据集，这种算法效果相当好。

- alpha：一个浮点数。该参数给出了正则化项的系数。
- max_iter：一个整数。该参数指定了最大迭代次数。一旦到达该迭代数量或者算法成功收敛，则迭代终止。
- tol：一个浮点数。该参数指定了最优化过程的收敛性阈值。该阈值用于判断收敛条件。
- learning_rate_init：一个小数。该参数为初始学习率。只有当使用sgd算法或者adam算法时，该参数才有意义。
- verbose：一个布尔值。如果为True则输出算法中间信息（用于跟踪调试）。
- early_stopping：一个布尔值。如果为True，则启用早停策略。只有当使用sgd算法或者adam算法时，该参数才有意义。
  - 如果启动了早停策略，则算法自动将 validation_fraction 的数据用作验证数据集用的验证。
- validation_fraction：一个浮点数，必须位于 0 和 1 之间。如果启用了启动策略，则划分训练数据集的该份额用于验证集。其重要的属性如下。
- classes_：每个输出的类别。
- loss_：当前损失函数值。
- coefs_：给出权值。
- intercepts_：给出阈值。
- n_iter：给出实际迭代次数。

8.3.6 多层神经网络与线性不可分数据集

下面为我们在线性不可分数据集上使用多层神经网络的情形。为了便于观察结果，我们使用二维的特征数据。首先是生成线性不可分数据集的方法：

```
def creat_data_no_linear_2d(n):
        np.random.seed(1)
        x_11=np.random.randint(0,100,(n,1))
        x_12=10+np.random.randint(-5,5,(n,1,))
        x_21=np.random.randint(0,100,(n,1))
        x_22=20+np.random.randint(0,10,(n,1))

        x_31=np.random.randint(0,100,(int(n/10),1))
        x_32=20+np.random.randint(0,10,(int(n/10),1))

        new_x_11=x_11*np.sqrt(2)/2-x_12*np.sqrt(2)/2## 沿X轴旋转45°
        new_x_12=x_11*np.sqrt(2)/2+x_12*np.sqrt(2)/2## 沿X轴旋转45°
        new_x_21=x_21*np.sqrt(2)/2-x_22*np.sqrt(2)/2## 沿X轴旋转45°
        new_x_22=x_21*np.sqrt(2)/2+x_22*np.sqrt(2)/2## 沿X轴旋转45°
        new_x_31=x_31*np.sqrt(2)/2-x_32*np.sqrt(2)/2## 沿X轴旋转45°
        new_x_32=x_31*np.sqrt(2)/2+x_32*np.sqrt(2)/2## 沿X轴旋转45°
```

```
    plus_samples=np.hstack([new_x_11,new_x_12,np.ones((n,1))])
    minus_samples=np.hstack([new_x_21,new_x_22,-np.ones((n,1))])
    err_samples=np.hstack([new_x_31,new_x_32,np.ones((int(n/10),1))])
    samples=np.vstack([plus_samples,minus_samples,err_samples])
    np.random.shuffle(samples)  # 混洗数据
    return samples
```

- 参数：n负类实例的个数。正类实例的个数为 $1.1\times n$。
- 返回值：所有的样本点组成的数组，形状为 (2*n,3)。数组中每一行代表一个样本点，由其特征 $\vec{x}$ 和标记 y 组成。

其过程为：首先在 y 轴坐标为 5~15 之间生成 n 个随机点作为正类，在 y 轴坐标为 20 的上方生成 n 个随机点作为负类。再在 y 轴坐标为 20~30 之间生成 $n/10$ 个随机点作为正类。然后 45° 旋转 x 坐标轴，再返回这些点在新坐标轴中的坐标。注意这里混洗了数据，否则会发现数据集的前半部分都是正类，后半部分都是负类，需要混洗数据从而让正负类交叉出现。

然后给出绘制线性不可分数据集的方法：

```
def plot_samples_2d(ax,samples):
        Y=samples[:,-1]
        position_p=Y==1 ## 正类位置
        position_m=Y==-1 ## 负类位置
        ax.scatter(samples[position_p,0],samples[position_p,1],
            marker='+',label='+',color='b')
        ax.scatter(samples[position_m,0],samples[position_m,1],
            marker='^',label='-',color='y')
```

- 参数
  - ax：一个 Axes实例，负责绘制图形。
  - samples：代表训练数据集的数组，形状为 (N,n_features+1)，其中N为样本点的个数，n_features代表特征数量（这里为 2，表示两个特征）。

该函数的用法为：

```
fig=plt.figure()
ax=fig.add_subplot(1,1,1)
data=creat_data_no_linear_2d(100)
plot_samples_2d(ax,data)
ax.legend(loc='best')
plt.show()
```

其图形如图 8.11 所示。

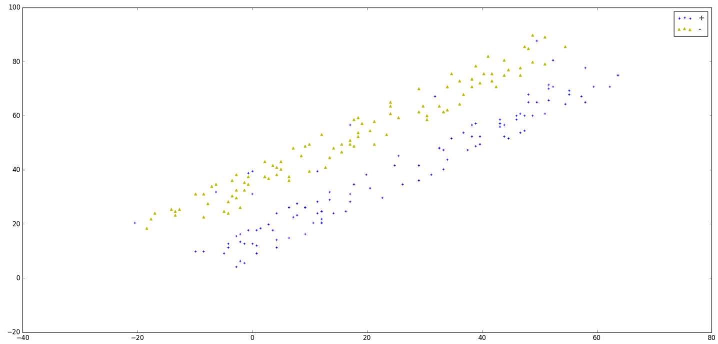


图 8.11 perceptron_no_linear_2d

用多层神经网络MLPClassifier来处理非线性数据集。函数为：

```
def predict_with_MLPClassifier(ax,train_data):
        train_x=train_data[:,:-1]
        train_y=train_data[:,-1]
        clf=MLPClassifier(activation='logistic',max_iter=1000)# 构造分类器实例
        clf.fit(train_x,train_y) # 训练分类器
        print(clf.score(train_x,train_y)) # 查看在训练集上的评价预测精度

        ## 用训练好的训练集预测平面上每一点的输出##
        x_min, x_max = train_x[:, 0].min() - 1, train_x[:, 0].max() + 2
        y_min, y_max = train_x[:, 1].min() - 1, train_x[:, 1].max() + 2
        plot_step=1
        xx, yy = np.meshgrid(np.arange(x_min, x_max, plot_step),
            np.arange(y_min, y_max, plot_step))
        Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
        Z = Z.reshape(xx.shape)
        ax.contourf(xx, yy, Z, cmap=plt.cm.Paired)
```

其过程为：首先用训练数据集数据来训练多层神经网络分类器。然后用分类器预测平面上每个点的输出并绘制轮廓图。

最后使用predict_with_MLPClassifier函数：

```
data=creat_data_no_linear_2d(500)
fig=plt.figure()
ax=fig.add_subplot(1,1,1)
predict_with_MLPClassifier(ax,data)
plot_samples_2d(ax,data)
ax.legend(loc='best')
plt.show()
```

- 输出的准确率为：0.952380952381。
- 输出图形如图 8.12 所示。

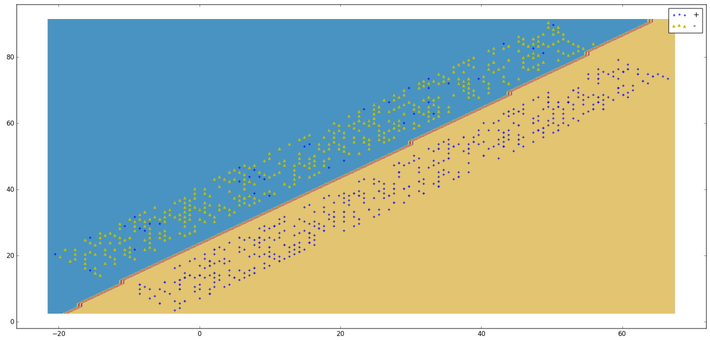


图 8.12 MLPClassifier_no_linear

8.3.7 多层神经网络的应用

最后来看一个真实的例子：对鸢尾花进行分类。鸢尾花数据集一共有 150 个数据，这些数据分为 3 类（分别为setosa，versicolor，virginica），每类 50 个数据。每个数据包含 4 个属性：萼片 (sepal) 长度、萼片宽度、花瓣 (petal) 长度、花瓣宽度。

首先我们加载数据：

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import neighbors, datasets
np.random.seed(0)
iris=load_iris()
X=iris.data[:,0:2]
Y=iris.target
data=np.hstack((X,Y.reshape(Y.size,1)))
np.random.shuffle(data)
X=data[:,:-1]
Y=data[:,-1]
train_x=X[:-30]
test_x=X[-30:]
train_y=Y[:-30]
test_y=Y[-30:]
```

在这里X=iris.data[:,0:2]表示使用的是sepal length和sepal width这两个特征。因为两个特征可以方便地在二维图形上表现出来，横轴为sepal length，纵轴为sepal width。

然后对数据进行混洗。因为原始数据是这样的：前 50 个样本点属于类别 1，中间 50 个样本点属于类别 2，最后 50 个样本点属于类别 3。为了更好地进行训练神经网络，将数据顺序随机重排。

最后取出最后的 30 个数据作为测试集。可以看到如果不重新排列样本，则最后 30 个数据肯定全部是属于类别 2 的。

给出测试函数如下：

```
def mlpclassifier_iris():
        fig=plt.figure()
        ax=fig.add_subplot(1,1,1)
        classifier=MLPClassifier(activation='logistic',max_iter=10000,
            hidden_layer_sizes=(30,))
        classifier.fit(train_x,train_y)
        train_score=classifier.score(train_x,train_y)
        test_score=classifier.score(test_x,test_y)
        x_min, x_max = train_x[:, 0].min() - 1, train_x[:, 0].max() + 2
        y_min, y_max = train_x[:, 1].min() - 1, train_x[:, 1].max() + 2
        plot_classifier_predict_meshgrid(ax,classifier,x_min,x_max,y_min,y_max)
        plot_samples(ax,train_x,train_y)
        ax.legend(loc='best')
        ax.set_xlabel(iris.feature_names[0])
        ax.set_ylabel(iris.feature_names[1])
        ax.set_title("train score:%f;test score:%f"%(train_score,test_score))
        plt.show()
```

其中使用了两个绘图函数：

- ❑ plot_samples：用于绘制样本点

```
def plot_samples(ax,x,y):
    n_classes = 3
    plot_colors = "bry"
    for i, color in zip(range(n_classes), plot_colors):
        idx = np.where(y == i)
        ax.scatter(x[idx, 0], x[idx, 1], c=color,
            label=iris.target_names[i], cmap=plt.cm.Paired)
```

  - ❍ 参数
    - ❖ ax：一个 Axes实例，用于绘制图形。
    - ❖ x：样本点的第一个特征。
    - ❖ y：样本点的第二个特征。

- ❑ plot_classifier_predict_meshgrid：用于绘制分类器对平面上每个点进行分类预测的分布图。

```
def plot_classifier_predict_meshgrid(ax,clf,x_min,x_max,y_min,y_max):
    plot_step = 0.02
    xx, yy = np.meshgrid(np.arange(x_min, x_max, plot_step),
        np.arange(y_min, y_max, plot_step))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    ax.contourf(xx, yy, Z, cmap=plt.cm.Paired)
```

❍ 参数

❖ ax：一个 Axes实例，用于绘制图形。

❖ clf：一个分类器。

❖ x_min：样本点的第一个特征取值的下界。

❖ x_max：样本点的第一个特征取值的上界。

❖ y_min：样本点的第二个特征取值的下界。

❖ y_max：样本点的第二个特征取值的上界。

然后调用mlpclassifier_iris函数，结果如图 8.13 所示。其中score值表示预测精度（图 8.13 表示：分类器在训练数据上的的预测精度为 80.8333%，在测试集上的预测精度为 80.0%）。

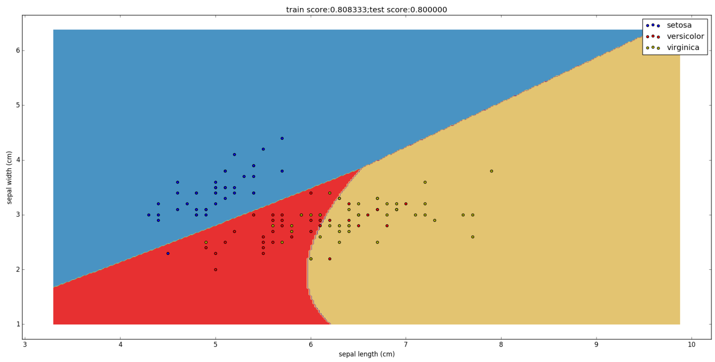


图 8.13 MLPClassifier_iris

下面观察不同的隐含层的对于多层神经网络分类器的影响：

```
def mlpclassifier_iris_hidden_layer_sizes():
        fig=plt.figure()
        hidden_layer_sizes=[(10,),(30,),(100,),(5,5),(10,10),(30,30)]
        for itx,size in enumerate(hidden_layer_sizes):
            ax=fig.add_subplot(2,3,itx+1)
            classifier=MLPClassifier(activation='logistic',max_iter=10000
                ,hidden_layer_sizes=size)
            classifier.fit(train_x,train_y)
            train_score=classifier.score(train_x,train_y)
            test_score=classifier.score(test_x,test_y)
            x_min, x_max = train_x[:, 0].min() - 1, train_x[:, 0].max() + 2
            y_min, y_max = train_x[:, 1].min() - 1, train_x[:, 1].max() + 2
            plot_classifier_predict_meshgrid(ax,classifier,x_min,x_max,y_min,y_max)
            plot_samples(ax,train_x,train_y)
            ax.legend(loc='best')
            ax.set_xlabel(iris.feature_names[0])
            ax.set_ylabel(iris.feature_names[1])
            ax.set_title("layer_size:%s;train score:%f;test score:%f"
                %(size,train_score,test_score))
        plt.show()
```

调用mlpclassifier_iris_hidden_layer_sizes函数。结果如图 8.14 所示。可以看到，由于总的数据集样本数量仅为 150 个，因此当神经网络的功能单元过多（比如 100 个）时，训练精度和预测精度便急剧下降。但是另一方面，对于神经网络的隐含层为 (5,5)结构时，训练精度和预测精度也比较差。正如前文所说，如何设置隐含层神经元的个数理论上尚未解决，实际中靠“试错法”来调整。

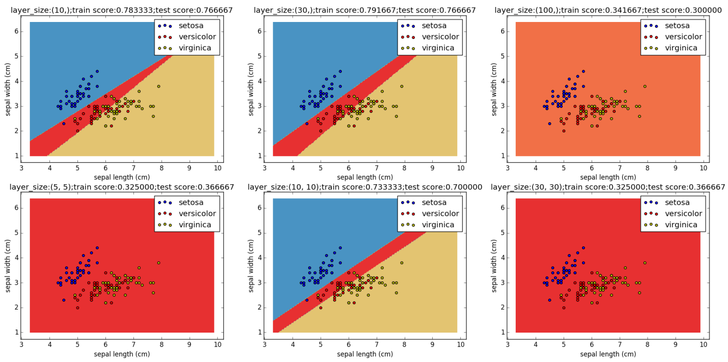


图 8.14　MLPClassifier_iris_layer

下面观察激活函数对于多层神经网络分类器的影响：

```
def mlpclassifier_iris_ativations():
        fig=plt.figure()
        ativations=["logistic","tanh","relu"]
        for itx,act in enumerate(ativations):
            ax=fig.add_subplot(1,3,itx+1)
            classifier=MLPClassifier(activation=act,max_iter=10000,
                hidden_layer_sizes=(30,))
            classifier.fit(train_x,train_y)
            train_score=classifier.score(train_x,train_y)
            test_score=classifier.score(test_x,test_y)
            x_min, x_max = train_x[:, 0].min() - 1, train_x[:, 0].max() + 2
            y_min, y_max = train_x[:, 1].min() - 1, train_x[:, 1].max() + 2
            plot_classifier_predict_meshgrid(ax,classifier,x_min,x_max,y_min,y_max)
            plot_samples(ax,train_x,train_y)
            ax.legend(loc='best')
            ax.set_xlabel(iris.feature_names[0])
            ax.set_ylabel(iris.feature_names[1])
            ax.set_title("activation:%s;train score:%f;test score:%f"
                %(act,train_score,test_score))
        plt.show()
```

调用mlpclassifier_iris_ativations函数，结果如图 8.15 所示。可以看到，不同的激活函数对性能有影响，但是相互之间没有显著差别。

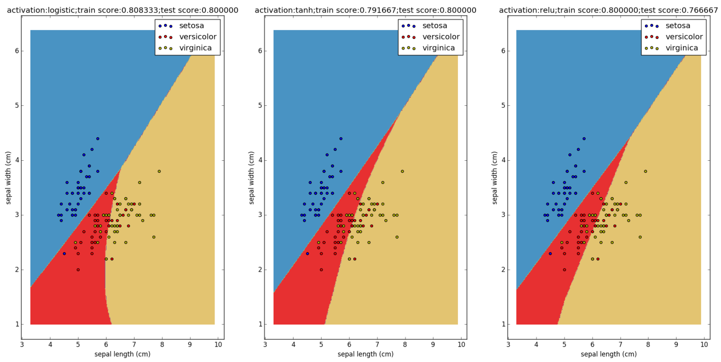


图 8.15 MLPClassifier_iris_activation

下面观察优化算法对于多层神经网络分类器的影响：

```
def mlpclassifier_iris_algorithms():
        fig=plt.figure()
        algorithms=["l-bfgs","sgd","adam"]
        for itx,algo in enumerate(algorithms):
            ax=fig.add_subplot(1,3,itx+1)
            classifier=MLPClassifier(activation="tanh",max_iter=10000,
                hidden_layer_sizes=(30,),algorithm=algo)
            classifier.fit(train_x,train_y)
            train_score=classifier.score(train_x,train_y)
            test_score=classifier.score(test_x,test_y)
            x_min, x_max = train_x[:, 0].min() - 1, train_x[:, 0].max() + 2
            y_min, y_max = train_x[:, 1].min() - 1, train_x[:, 1].max() + 2
            plot_classifier_predict_meshgrid(ax,classifier,x_min,x_max,y_min,y_max)
            plot_samples(ax,train_x,train_y)
            ax.legend(loc='best')
            ax.set_xlabel(iris.feature_names[0])
            ax.set_ylabel(iris.feature_names[1])
            ax.set_title("algorithm:%s;train score:%f;test score:%f"
                %(algo,train_score,test_score))
     plt.show()
```

调用mlpclassifier_iris_algorithms函数，结果如图 8.16 所示。可以看到，不同的优化算法对性能有重要影响，但是相互之间没有显著差别。

最后观察学习 η 对多层神经网络分类器的影响：

```
def mlpclassifier_iris_eta():
        fig=plt.figure()
        etas=[0.1,0.01,0.001,0.0001]
        for itx,eta in enumerate(etas):
            ax=fig.add_subplot(2,2,itx+1)
            classifier=MLPClassifier(activation="tanh",max_iter=1000000,
```

```
    hidden_layer_sizes=(30,),algorithm='sgd',learning_rate_init=eta)
    classifier.fit(train_x,train_y)
    iter_num=classifier.n_iter_
    train_score=classifier.score(train_x,train_y)
    test_score=classifier.score(test_x,test_y)
    x_min, x_max = train_x[:, 0].min() - 1, train_x[:, 0].max() + 2
    y_min, y_max = train_x[:, 1].min() - 1, train_x[:, 1].max() + 2
    plot_classifier_predict_meshgrid(ax,classifier,x_min,x_max,y_min,y_max)
    plot_samples(ax,train_x,train_y)
    ax.legend(loc='best')
    ax.set_xlabel(iris.feature_names[0])
    ax.set_ylabel(iris.feature_names[1])
    ax.set_title("eta:%f;train score:%f;test score:%f;iter_num:%d"
        %(eta,train_score,test_score,iter_num))
plt.show()
```

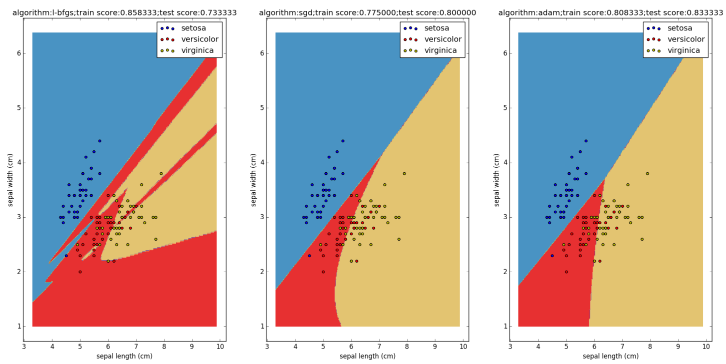


图 8.16 MLPClassifier_iris_algorithm

调用mlpclassifier_iris_eta函数，结果如图 8.17 所示。可以看到，如果采用随机梯度下降法，则随着 η 的减小，总体的迭代次数（iter_num）会增加；但是并不是 η 越小预测性能越好。在 η 过小时，预测精度反而会下降。

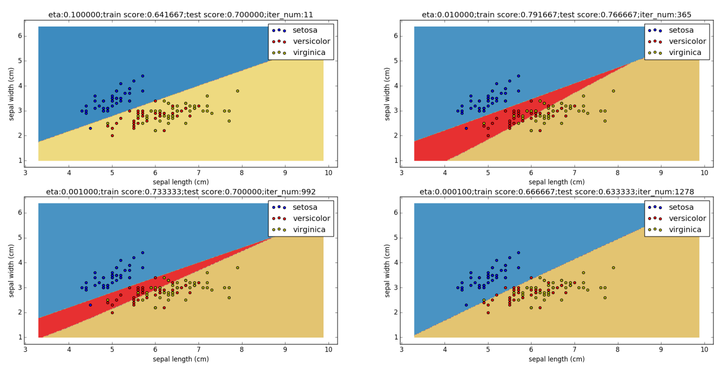


图 8.17 MLPClassifier_iris_eta

第 9 章

半监督学习

9.1 概述

在机器学习领域，监督学习 (Supervised learning)、非监督学习 (Unsupervised learning) 以及半监督学习 (Semi-supervised learning) 是三类研究比较多，应用比较广的学习技术。

监督学习：通过已有的一部分输入数据与输出数据之间的对应关系，生成一个函数，将输入映射到合适的输出，例如分类。

非监督学习：直接对输入数据集进行建模，例如聚类。

半监督学习：综合利用有类标的数据和没有类标的数据，来生成合适的分类函数。

现实应用中往往能够容易地收集到大量的未标记样本。假设将训练数据集 D 分成两个部分 $D = D_l \bigcup D_u$。其中 $D_l = \{(\vec{x}_1, y_1), (\vec{x}_2, y_2), \cdots, (\vec{x}_l, y_l)\}$ 包含了 l 个有标记的样本；而 $D_u = \{(\vec{x}_{l+1}, y_{l+1}), (\vec{x}_{l+2}, y_{l+2}), \cdots, (\vec{x}_{l+u}, y_{l+u})\}$ 包含了 u 个未标记样本，其中 $l \ll u$。

如果只是简单地利用标记样本数据集 D_l 来学习，则浪费了 D_u 中的大量数据。事实上 D_u 虽然没有直接包含标记信息，但是如果假设它与 D_l 是从同样的数据源独立分步采样而来的，则可以获得一些有效信息。半监督学习`semi-supervised learning` 就是一类如何自动地利用未标记的 D_u 来提升学习性能的算法。

9.2 算法笔记精华

9.2.1 生成式半监督学习方法

生成式（generative methods）半监督学习方法是一类基于生成式模型的算法。根据假设的生成式模型的不同，将产生不同的方法。这里以生成式高斯混合模型为例，讲解其原理和求解方法。

此类方法假设：训练数据集 D 中的样本都是由同一个潜在的模型生成的。该模型就是生成式模型（所谓生成，就是样本的生成）。

生成式高斯混合模型原理

假设类别标记的取值空间为 $\mathcal{Y}=\{1,2,\cdots,K\}$ 。假设样本由高斯混合模型产生，且每个类别对应一个高斯混合成分。令数据样本的概率密度为：

$$p(\vec{\boldsymbol{x}})=\sum_{k=1}^{K}\alpha_k p_k(\vec{\boldsymbol{x}};\vec{\mu}_k,\Sigma_k)$$

其中 $p_k(\vec{\boldsymbol{x}};\vec{\mu}_k,\Sigma_k)$ 是第 k 个高斯混合成分的概率，$\vec{\mu}_k,\Sigma_k$ 为该高斯混合成分的参数。混合系数 $\alpha_k\geqslant 0,\sum_{k=1}^{K}\alpha_k=1$。

设模型 f 对 $\vec{\boldsymbol{x}}$ 的预测标记为 $f(\vec{\boldsymbol{x}})\in\mathcal{Y}$。令 $\Theta\in\{1,2,\cdots,K\}$ 表示样本 $\vec{\boldsymbol{x}}$ 所属的高斯混合成分。根据最大化后验概率，有：

$$f(\vec{\boldsymbol{x}})=\arg\max_{j\in\mathcal{Y}}p(y=j/\vec{\boldsymbol{x}})$$

考虑到 $p(y=j/\vec{\boldsymbol{x}})=\sum_{k=1}^{K}p(y=j,\Theta=k/\vec{\boldsymbol{x}})$，则有：

$$f(\vec{\boldsymbol{x}})=\arg\max_{j\in\mathcal{Y}}\sum_{k=1}^{K}p(y=j,\Theta=k/\vec{\boldsymbol{x}})$$

由于 $p(y=j,\Theta=k/\vec{\boldsymbol{x}})=p(y=j/\Theta=k,\vec{\boldsymbol{x}})\cdot p(\Theta=k/\vec{\boldsymbol{x}})$，则有：

$$f(\vec{\boldsymbol{x}})=\arg\max_{j\in\mathcal{Y}}\sum_{k=1}^{K}p(y=j/\Theta=k,\vec{\boldsymbol{x}})\cdot p(\Theta=k/\vec{\boldsymbol{x}})$$

$p(\Theta=k/\vec{\boldsymbol{x}})$ 为样本 $\vec{\boldsymbol{x}}$ 由第 k 个高斯混合成分生成的后验概率，$p(y=j/\Theta=k,\vec{\boldsymbol{x}})$ 为 $\vec{\boldsymbol{x}}$ 由第 k 个高斯混合成分生成，且其类别为 j 的概率。根据定义有：

$$p(\Theta = k/\vec{\boldsymbol{x}}) = \frac{\alpha_k p_k(\vec{\boldsymbol{x}}; \vec{\mu}_k, \Sigma_k)}{\sum_{k=1}^{K} \alpha_k p_k(\vec{\boldsymbol{x}}; \vec{\mu}_k, \Sigma_k)}$$

在公式

$$f(\vec{\boldsymbol{x}}) = \arg\max_{j \in \mathcal{Y}} \sum_{k=1}^{K} p(y = j/\Theta = k, \vec{\boldsymbol{x}}) \cdot p(\Theta = k/\vec{\boldsymbol{x}})$$

中，$p(y = j/\Theta = k, \vec{\boldsymbol{x}})$ 需要知道样本 $\vec{\boldsymbol{x}}$ 的标记 y；而 $p(\Theta = k/\vec{\boldsymbol{x}})$ 并不需要样本的标记，仅提供未标记样本即可。因此提供大量的未标记数据可以更加准确地估计 $p(\Theta = k/\vec{\boldsymbol{x}})$，从而提高模型的整体预测性能。

生成式高斯混合模型求解

生成式高斯混合模型通过EM算法求解。给定标记样本集 $D_l = \{(\vec{\boldsymbol{x}}_1, y_1), (\vec{\boldsymbol{x}}_2, y_2), \cdots, (\vec{\boldsymbol{x}}_l, y_l)\}$，和未标记样本集 $D_u = \{\vec{\boldsymbol{x}}_{l+1}, \vec{\boldsymbol{x}}_{l+2}, \cdots, \vec{\boldsymbol{x}}_{l+u}\}$，其中 $l \ll u$，$l + u = N$。待求的参数为 $\{(\alpha_k, \vec{\mu}_k, \Sigma_k) \mid k = 1, 2, \cdots, K\}$。

首先给出 $D_l \bigcup D_u$ 的对数似然函数：

$$L(D_l \bigcup D_u) = \sum_{(\vec{\boldsymbol{x}}_i, y_i) \in D_l} \log\left(\sum_{k=1}^{K} \alpha_k p_k(\vec{\boldsymbol{x}}_i; \vec{\mu}_k, \Sigma_k) \cdot p(y_i/\Theta = k, \vec{\boldsymbol{x}}_i)\right) + \sum_{\vec{\boldsymbol{x}}_i \in D_u} \log\left(\sum_{k=1}^{K} \alpha_k p_k(\vec{\boldsymbol{x}}_i; \vec{\mu}_k, \Sigma_k)\right)$$

上式中：

- ❑ 第一个对数项中，为联合概率 $p(\vec{\boldsymbol{x}}_i, y_i)$:

$$p(\vec{\boldsymbol{x}}_i, y_i) = p(y_i/\vec{\boldsymbol{x}}_i) p(\vec{\boldsymbol{x}}_i) = \sum_{k=1}^{K} \alpha_k p_k(\vec{\boldsymbol{x}}_i; \vec{\mu}_k, \Sigma_k) \cdot p(y_i/\Theta = k, \vec{\boldsymbol{x}}_i)$$

- ❑ 第二个对数项中，为概率 $p(\vec{\boldsymbol{x}}_i)$。

然后用EM算法求解。在EM算法中，迭代更新步骤如下。

- ❑ E步：根据当前模型参数计算未标记样本 $\vec{\boldsymbol{x}}_i$ 属于各高斯混合成分的概率：

$$\gamma_{i,k} = \frac{\alpha_k p_k(\vec{\boldsymbol{x}}_i; \vec{\mu}_k, \Sigma_k)}{\sum_{k=1}^{K} \alpha_k p_k(\vec{\boldsymbol{x}}_i; \vec{\mu}_k, \Sigma_k)}$$

- ❑ M步：基于 $\gamma_{i,k}$ 更新模型参数。令 l_k 为第 k 类的有标记样本数目，则：

$$\vec{\mu}_k = \frac{1}{\sum_{\vec{x}_i \in D_u} \gamma_{i,k} + l_k} \left(\sum_{\vec{x}_i \in D_u} \gamma_{i,k} \vec{x}_i + \sum_{(\vec{x}_i, y_i) \in D_l \textit{ and } y_i = k} \gamma_{i,k} \vec{x}_i \right)$$

$$\begin{aligned} \Sigma_k = & \frac{1}{\sum_{\vec{x}_i \in D_u} \gamma_{i,k} + l_k} \\ & \times \left(\sum_{\vec{x}_i \in D_u} \gamma_{i,k} (\vec{x}_i - \vec{\mu}_k)(\vec{x}_i - \vec{\mu}_k)^T + \sum_{(\vec{x}_i, y_i) \in D_l \textit{ and } y_i = k} \gamma_{i,k} (\vec{x}_i - \vec{\mu}_k)(\vec{x}_i - \vec{\mu}_k)^T \right) \end{aligned}$$

$$\alpha_k = \frac{1}{N} \left(\sum_{\vec{x}_i \in D_u} \gamma_{i,k} + l_k \right)$$

上述过程不断迭代直至收敛为止。然后根据式子：

$$f(\vec{x}) = \arg\max_{j \in \mathcal{Y}} \sum_{k=1}^{K} p(y = j / \Theta = k, \vec{x}) \cdot p(\Theta = k / \vec{x})$$

$$p(\Theta = k / \vec{x}) = \frac{\alpha_k p_k(\vec{x}; \vec{\mu}_k, \Sigma_k)}{\sum_{k=1}^{K} \alpha_k p_k(\vec{x}; \vec{\mu}_k, \Sigma_k)}$$

来对样本进行分类。

如果将高斯混合模型替换成其他模型，则可以推导出其他模型的生成式半监督学习方法。生成式半监督学习方法的优缺点如下。

- 优点：方法简单，容易实现。通常在有标记数据极少时，生成式半监督学习方法比其他方法性能更好。
- 缺点：假设的生成式模型必须与真实数据分布吻合。如果不吻合则可能效果很差。而如何给出与真实数据分布吻合的生成式模型，这就需要对问题领域的充分了解。

9.2.2 图半监督学习

可以用图 $G(V, E)$ 来表示一个样本集 D。

- 顶点 V ：代表每个样本 $\vec{x}_i$。
- 边 E ：代表样本之间的相似度。若两个样本之间的相似度很高，则对应的顶点之间存在一条边。边的权重正比于样本之间的相似度。

将所有的标记样本对应的顶点，根据其标记染色为不同的颜色；而未标记样本所对应的顶点待染色。半监督学习的目标就是将所有这些未标记样本也染色。这就是“颜色”在图上传播的过程，称为标记传播算法`label propagation`。

标记传播算法原理

给定标记样本集 $D_l=\{(\vec{\boldsymbol{x}}_1,y_1),(\vec{\boldsymbol{x}}_2,y_2),\cdots,(\vec{\boldsymbol{x}}_l,y_l)\},y_i\in\{-1,+1\},i=1,2,\cdots,l$，和未标记样本集 $D_u=\{\vec{\boldsymbol{x}}_{l+1},\vec{\boldsymbol{x}}_{l+2},\cdots,\vec{\boldsymbol{x}}_{l+u}\}$，其中 $l\ll u,\quad l+u=N$。首先基于 $D_l\bigcup D_u$ 构建一个图 $G=(V,E)$。其中

- 顶点集 $V=\{\vec{\boldsymbol{x}}_1,\vec{\boldsymbol{x}}_2,\cdots,\vec{\boldsymbol{x}}_l,\vec{\boldsymbol{x}}_{l+1},\vec{\boldsymbol{x}}_{l+2},\cdots,\vec{\boldsymbol{x}}_{l+u}\}$。
- 边集 E 用矩阵表示（称为亲和矩阵 `affinity matirx`，常基于高斯函数）：

$$(\boldsymbol{W})_{i,j}=\begin{cases}\exp\left(-\frac{\|\vec{\boldsymbol{x}}_i-\vec{\boldsymbol{x}}_j\|_2^2}{2\sigma^2}\right), & \text{if}\quad i\neq j\\ 0, & \text{otherwise}\end{cases}$$
$$i,j\in\{1,2,\cdots,N\}$$

其中 $\sigma>0$ 是用户指定的参数。可以看到 $(\boldsymbol{W})_{i,j}=(\boldsymbol{W})_{j,i}$, $\boldsymbol{W}$ 为对称矩阵。

我们的目标是：从图 $G=(V,E)$ 将学得一个实值函数 $f:V\rightarrow\mathbb{R}$，其对应的分类规则为：$y_i=\text{sign}(f(\vec{\boldsymbol{x}}_i)),y_i\in\{-1,+1\}$。

假设：相似的样本应该具有相似的标记。于是，定义关于 f 的能量函数`energy function`：

$$\begin{aligned}E(f)&=\frac{1}{2}\sum_{i=1}^N\sum_{j=1}^N(\boldsymbol{W})_{i,j}\left(f(\vec{\boldsymbol{x}}_i)-f(\vec{\boldsymbol{x}}_j)\right)^2\\&=\frac{1}{2}\sum_{i=1}^N\sum_{j=1}^N\left[(\boldsymbol{W})_{i,j}f(\vec{\boldsymbol{x}}_i)^2+(\boldsymbol{W})_{i,j}f(\vec{\boldsymbol{x}}_j)^2-2(\boldsymbol{W})_{i,j}f(\vec{\boldsymbol{x}}_i)f(\vec{\boldsymbol{x}}_j)\right]\\&=\frac{1}{2}\left[\sum_{i=1}^N\left(f(\vec{\boldsymbol{x}}_i)^2\sum_{j=1}^N(\boldsymbol{W})_{i,j}\right)+\sum_{j=1}^N\left(f(\vec{\boldsymbol{x}}_j)^2\sum_{i=1}^N(\boldsymbol{W})_{i,j}\right)-2\sum_{i=1}^N\sum_{j=1}^N(\boldsymbol{W})_{i,j}f(\vec{\boldsymbol{x}}_i)f(\vec{\boldsymbol{x}}_j)\right]\end{aligned}$$

其定义如下。

- 对角矩阵 $\boldsymbol{D}=\text{diag}(d_1,d_2,\cdots,d_N)$，其中 $d_i=\sum_{j=1}^N(\boldsymbol{W})_{i,j}$ 为矩阵 $\boldsymbol{W}$ 的第 i 行元素之和。

$$\boldsymbol{D}=\begin{bmatrix}d_1&0&0&\cdots&0\\0&d_2&0&\cdots&0\\0&0&d_3&\cdots&0\\\vdots&\vdots&\vdots&\ddots&\vdots\\0&0&0&\cdots&d_N\end{bmatrix}$$

- $\vec{\boldsymbol{f}}_l=(f(\vec{\boldsymbol{x}}_1),f(\vec{\boldsymbol{x}}_2),\cdots,f(\vec{\boldsymbol{x}}_l))^T$, $\vec{\boldsymbol{f}}_u=(f(\vec{\boldsymbol{x}}_{l+1}),f(\vec{\boldsymbol{x}}_{l+2}),\cdots,f(\vec{\boldsymbol{x}}_{l+u}))^T$ 分别为函数 f 在有标记样本和未标记样本上的预测结果。定义 $\vec{\boldsymbol{f}}=(f(\vec{\boldsymbol{x}}_1),\cdots,f(\vec{\boldsymbol{x}}_l),f(\vec{\boldsymbol{x}}_{l+1}),\cdots,f(\vec{\boldsymbol{x}}_{l+u}))^T$

结合 $\boldsymbol{D}$ 的定义以及 $\boldsymbol{W}$ 的对称性，有：

$$
\begin{aligned}
E(f) &= \frac{1}{2}\left[\sum_{i=1}^{N} f(\vec{\boldsymbol{x}}_i)^2 d_i + \sum_{j=1}^{N} f(\vec{\boldsymbol{x}}_j)^2 d_j - 2\sum_{i=1}^{N}\sum_{j=1}^{N}(\boldsymbol{W})_{i,j} f(\vec{\boldsymbol{x}}_i) f(\vec{\boldsymbol{x}}_j)\right] \\
&= \sum_{i=1}^{N} f(\vec{\boldsymbol{x}}_i)^2 d_i - \sum_{i=1}^{N}\sum_{j=1}^{N}(\boldsymbol{W})_{i,j} f(\vec{\boldsymbol{x}}_i) f(\vec{\boldsymbol{x}}_j) \\
&= \vec{\boldsymbol{f}}^T(\boldsymbol{D}-\boldsymbol{W})\vec{\boldsymbol{f}}
\end{aligned}
$$

标记传播算法求解

我们需要找出具有最小能量的函数 f ，且在标记样本上满足 $f(\vec{\boldsymbol{x}}_i) = y_i, i = 1, 2, \cdots, l$；在未标记样本上满足 $(\boldsymbol{D}-\boldsymbol{W})\vec{\boldsymbol{f}} = \vec{\boldsymbol{0}}$。

以第 l 行第 l 列为界，采用分块矩阵表示方式：

$$
\boldsymbol{W} = \begin{bmatrix} \boldsymbol{W}_{l,l} & \boldsymbol{W}_{l,u} \\ \boldsymbol{W}_{u,l} & \boldsymbol{W}_{u,u} \end{bmatrix}, \quad \boldsymbol{D} = \begin{bmatrix} \boldsymbol{D}_{l,l} & \boldsymbol{0}_{l,u} \\ \boldsymbol{0}_{u,l} & \boldsymbol{D}_{u,u} \end{bmatrix}
$$

则：

$$
\begin{aligned}
E(f) &= \vec{\boldsymbol{f}}^T(\boldsymbol{D}-\boldsymbol{W})\vec{\boldsymbol{f}} \\
&= (\vec{\boldsymbol{f}}_l^T \quad \vec{\boldsymbol{f}}_u^T)\left(\begin{bmatrix} \boldsymbol{D}_{l,l} & \boldsymbol{0}_{l,u} \\ \boldsymbol{0}_{u,l} & \boldsymbol{D}_{u,u} \end{bmatrix} - \begin{bmatrix} \boldsymbol{W}_{l,l} & \boldsymbol{W}_{l,u} \\ \boldsymbol{W}_{u,l} & \boldsymbol{W}_{u,u} \end{bmatrix}\right)\begin{bmatrix} \vec{\boldsymbol{f}}_l \\ \vec{\boldsymbol{f}}_u \end{bmatrix} \\
&= \vec{\boldsymbol{f}}_l^T(\boldsymbol{D}_{l,l}-\boldsymbol{W}_{l,l})\vec{\boldsymbol{f}}_l - 2\vec{\boldsymbol{f}}_u^T\boldsymbol{W}_{u,l}\vec{\boldsymbol{f}}_l^T + \vec{\boldsymbol{f}}_u^T(\boldsymbol{D}_{u,u}-\boldsymbol{W}_{u,u})\vec{\boldsymbol{f}}_u
\end{aligned}
$$

为求得其最小值，由 $\frac{\partial E(f)}{\partial \vec{\boldsymbol{f}}_u} = \vec{\boldsymbol{0}}$ (因为 $\vec{\boldsymbol{f}}_l$ 是已知的)，得到：

$$
\vec{\boldsymbol{f}}_u = (\boldsymbol{D}_{u,u}-\boldsymbol{W}_{u,u})^{-1}\boldsymbol{W}_{u,l}\vec{\boldsymbol{f}}_l
$$

令:

$$
\begin{aligned}
\boldsymbol{P} = \boldsymbol{D}^{-1}\boldsymbol{W} &= \begin{bmatrix} \boldsymbol{D}_{l,l}^{-1} & \boldsymbol{0}_{l,u} \\ \boldsymbol{0}_{u,l} & \boldsymbol{D}_{u,u}^{-1} \end{bmatrix}\begin{bmatrix} \boldsymbol{W}_{l,l} & \boldsymbol{W}_{l,u} \\ \boldsymbol{W}_{u,l} & \boldsymbol{W}_{u,u} \end{bmatrix} \\
&= \begin{bmatrix} \boldsymbol{D}_{l,l}^{-1}\boldsymbol{W}_{l,l} & \boldsymbol{D}_{l,l}^{-1}\boldsymbol{W}_{l,u} \\ \boldsymbol{D}_{u,u}^{-1}\boldsymbol{W}_{u,l} & \boldsymbol{D}_{u,u}^{-1}\boldsymbol{W}_{u,u} \end{bmatrix}
\end{aligned}
$$

令：$\boldsymbol{P}_{u,u} = \boldsymbol{D}_{u,u}^{-1}\boldsymbol{W}_{u,u}$，　$\boldsymbol{P}_{u,l} = \boldsymbol{D}_{u,u}^{-1}\boldsymbol{W}_{u,l}$

则：

$$\begin{aligned}\vec{\boldsymbol{f}}_u &= (\boldsymbol{D}_{u,u} - \boldsymbol{W}_{u,u})^{-1}\boldsymbol{W}_{u,l}\vec{\boldsymbol{f}}_l \\ &= (\boldsymbol{D}_{u,u}(\boldsymbol{I} - \boldsymbol{D}_{u,u}^{-1}\boldsymbol{W}_{u,u}))^{-1}\boldsymbol{W}_{u,l}\vec{\boldsymbol{f}}_l \\ &= (\boldsymbol{I} - \boldsymbol{D}_{u,u}^{-1}\boldsymbol{W}_{u,u})^{-1}\boldsymbol{D}_{u,u}^{-1}\boldsymbol{W}_{u,l}\vec{\boldsymbol{f}}_l \\ &= (\boldsymbol{I} - \boldsymbol{P}_{u,u})^{-1}\boldsymbol{P}_{u,l}\vec{\boldsymbol{f}}_l\end{aligned}$$

于是，将 D_l 上的标记信息作为 $\vec{\boldsymbol{f}}_l = (y_1, y_2, \cdots, y_l)^T$ 代入，则得到未标记样本的预测值 $\vec{\boldsymbol{f}}_u$。

> $\boldsymbol{W}$ 由样本之间的相似程度决定，与标记无关。$\boldsymbol{D}$ 由 $\boldsymbol{W}$ 决定，$\boldsymbol{P}$ 也由 $\boldsymbol{W}$ 决定。

多类分类标记传播算法

前面给出的是二类分类问题的标记传播算法。下面介绍多类分类问题的标记传播算法。给定标记样本集 $D_l = \{(\vec{\boldsymbol{x}}_1, y_1), (\vec{\boldsymbol{x}}_2, y_2), \cdots, (\vec{\boldsymbol{x}}_l, y_l)\}, y_i \in \{1, 2, \cdots, K\}, i = 1, 2, \cdots, l$，和未标记样本集 $D_u = \{\vec{\boldsymbol{x}}_{l+1}, \vec{\boldsymbol{x}}_{l+2}, \cdots, \vec{\boldsymbol{x}}_{l+u}\}$，其中 $l \ll u,\quad l + u = N$。首先基于 $D_l \bigcup D_u$ 构建一个图 $G = (V, E)$。其中

- 节点集 $V = \{\vec{\boldsymbol{x}}_1, \vec{\boldsymbol{x}}_2, \cdots, \vec{\boldsymbol{x}}_l, \vec{\boldsymbol{x}}_{l+1}, \vec{\boldsymbol{x}}_{l+2}, \cdots, \vec{\boldsymbol{x}}_{l+u}\}$。
- 边集 E 用矩阵表示（称为亲和矩阵 `affinity matirx`，常基于高斯函数）：

$$(\boldsymbol{W})_{i,j} = \begin{cases}\exp\left(-\frac{||\vec{x}_i - \vec{x}_j||_2^2}{2\sigma^2}\right), & \text{if} \quad i \neq j \\ 0, & \text{otherwise}\end{cases}$$

$$i, j \in \{1, 2, \cdots, N\}$$

  其中 $\sigma > 0$ 是用户指定的高斯函数带宽参数。可以看到 $(\boldsymbol{W})_{i,j} = (\boldsymbol{W})_{j,i}$，$\boldsymbol{W}$ 为对称矩阵。
- 对角矩阵 $\boldsymbol{D} = \text{diag}(d_1, d_2, \cdots, d_N)$，其中 $d_i = \sum_{j=1}^N (\boldsymbol{W})_{i,j}$ 为矩阵 $\boldsymbol{W}$ 的第 i 行元素之和。

$$\boldsymbol{D} = \begin{bmatrix} d_1 & 0 & 0 & \cdots & 0 \\ 0 & d_2 & 0 & \cdots & 0 \\ 0 & 0 & d_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & d_N \end{bmatrix}$$

令 $\vec{\boldsymbol{x}}_i$ 的标记向量为 $\vec{\boldsymbol{F}}_i = (F_{i,1}, F_{i,2}, \cdots, F_{i,K})^T$，分类规则为：

$$y_i = \arg\max_{1 \leqslant j \leqslant K} F_{i,j}$$

定义非负的标记矩阵为：$\mathbf{F}=(\vec{\mathbf{F}}_1,\vec{\mathbf{F}}_2,\cdots,\vec{\mathbf{F}}_N)^T\in\mathbb{R}^{N\times K}$

$$\mathbf{F}=\begin{bmatrix}F_{1,1} & F_{1,2} & \cdots & F_{1,K}\\ F_{2,1} & F_{2,2} & \cdots & F_{2,K}\\ \vdots & \vdots & \ddots & \vdots\\ F_{N,1} & F_{N,2} & \cdots & F_{N,K}\end{bmatrix}$$

首先将 $\mathbf{F}$ 初始化为：

$$(\mathbf{F}^{<0>})_{i,j}=(\mathbf{Y})_{i,j}=\begin{cases}1, & \text{if}\quad 1\leqslant i\leqslant l\quad\text{and}\quad y_i=j\\ 0, & \text{otherwise}\end{cases}$$

即：$\mathbf{Y}$ 的前 l 行就是 l 个有标记样本的标记向量。其中 <0> 表示第 0 次迭代。

通过 $\mathbf{W}$ 构造一个标记传播矩阵 $\mathbf{S}=\mathbf{D}^{-1/2}\mathbf{W}\mathbf{D}^{-1/2}$。其中

$$\mathbf{D}^{-1/2}=\text{diag}(\frac{1}{\sqrt{d_1}},\frac{1}{\sqrt{d_2}},\cdots,\frac{1}{\sqrt{d_N}})=\begin{bmatrix}\frac{1}{\sqrt{d_1}} & 0 & 0 & \cdots & 0\\ 0 & \frac{1}{\sqrt{d_2}} & 0 & \cdots & 0\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ 0 & 0 & 0 & \cdots & \frac{1}{\sqrt{d_N}}\end{bmatrix}$$

则有：

$$\mathbf{F}^{<t+1>}=\alpha\mathbf{S}\mathbf{F}^{<t>}+(1-\alpha)\mathbf{Y}$$

其中 $\alpha\in(0,1)$ 为用户指定的参数，它是对标记传播项 $\mathbf{S}\mathbf{F}^{<t>}$ 和初始化项 $\mathbf{Y}$ 的重要性的一个折中。基于上式迭代直至收敛为止，于是：

$$\mathbf{F}^*=\lim_{t\to\infty}\mathbf{F}^{<t>}=(1-\alpha)(\mathbf{I}-\alpha\mathbf{S})^{-1}\mathbf{Y}$$

最后从 $\mathbf{F}^*$ 中可以获取 $\mathbb{D}_u$ 中样本的标记。

标记传播算法与正则化框架

上述算法都对应于如下正则化框架：

$$\min_{\mathbf{F}}\frac{1}{2}\left(\sum_{i,j=1}^{N}(\mathbf{W})_{i,j}||\frac{1}{\sqrt{d_i}}\vec{\mathbf{F}}_i-\frac{1}{\sqrt{d_j}}\vec{\mathbf{F}}_j||_2^2\right)+\mu\sum_{i=1}^{l}||\vec{\mathbf{F}}_i-\vec{\mathbf{Y}}_i||_2^2$$

其中：

- ❑ $\mu > 0$ 为正则化参数。当 $\mu = \frac{1-\alpha}{\alpha}$ 时，上式的最优解恰好为 $\boldsymbol{F}^* = (1-\alpha)(\boldsymbol{I} - \alpha\boldsymbol{S})^{-1}\boldsymbol{Y}$。
- ❑ 上式第一项为使得相近样本尽可能具有相似的标记。
- ❑ 上式第二项为使得学得结果在有标记样本上的预测与真实标记尽可能相同。

标记传播算法的优点是概念清晰。缺点是：存储开销较大，难以直接处理大规模数据；而且对于新的样本加入，需要对原图重构并进行标记传播。

迭代式标记传播算法

迭代式标记传播算法如下。

- ❑ 输入：
  - ❍ 有标记样本集 $D_l = \{(\vec{\boldsymbol{x}}_1, y_1), (\vec{\boldsymbol{x}}_2, y_2), \cdots, (\vec{\boldsymbol{x}}_l, y_l)\}, y_i \in \{1, 2, \cdots, K\}, i = 1, 2, \cdots, l$；
  - ❍ 未标记样本集 $D_u = \{\vec{\boldsymbol{x}}_{l+1}, \vec{\boldsymbol{x}}_{l+2}, \cdots, \vec{\boldsymbol{x}}_{l+u}\}, l + u = N$；
  - ❍ 构图参数 σ；
  - ❍ 折中参数 α。
- ❑ 输出：未标记样本的预测结果 $\hat{\vec{\boldsymbol{y}}} = (\hat{y}_{l+1}, \hat{y}_{l+2}, \cdots, \hat{y}_{l+u})^T, \hat{y}_i \in \{1, 2, \cdots, K\}, i = l+1, l+2, \cdots, l+u$。
- ❑ 步骤
  - ❍ 根据下式，计算 $\boldsymbol{W}$：

$$(\boldsymbol{W})_{i,j} = \begin{cases} \exp\left(-\frac{||\vec{\boldsymbol{x}}_i - \vec{\boldsymbol{x}}_j||_2^2}{2\sigma^2}\right), & \text{if} \quad i \neq j \\ 0, & \text{otherwise} \end{cases}$$

$$i, j \in \{1, 2, \cdots, N\}$$

  - ❍ 基于 $\boldsymbol{W}$ 构造标记传播矩阵 $\boldsymbol{S} = \boldsymbol{D}^{-1/2}\boldsymbol{W}\boldsymbol{D}^{-1/2}$。其中

$$\boldsymbol{D}^{-1/2} = \text{diag}\left(\frac{1}{\sqrt{d_1}}, \frac{1}{\sqrt{d_2}}, \cdots, \frac{1}{\sqrt{d_N}}\right) = \begin{bmatrix} \frac{1}{\sqrt{d_1}} & 0 & 0 & \cdots & 0 \\ 0 & \frac{1}{\sqrt{d_2}} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \frac{1}{\sqrt{d_N}} \end{bmatrix}$$

    $d_i = \sum_{j=1}^N (\boldsymbol{W})_{i,j}$ 为矩阵 $\boldsymbol{W}$ 的第 i 行元素之和。
  - ❍ 根据下式初始化 $\boldsymbol{F}^{<0>}$：

$$(\boldsymbol{F}^{<0>})_{i,j} = (\boldsymbol{Y})_{i,j} = \begin{cases} 1, & \text{if} \quad 1 \leqslant i \leqslant l \quad \text{and} \quad y_i = j \\ 0, & \text{otherwise} \end{cases}$$

- ❍ $t = 0$
- ❍ 迭代，迭代终止条件是 $\boldsymbol{F}$ 收敛至 $\boldsymbol{F}^*$。迭代过程为：
  - ❖ $\boldsymbol{F}^{<t+1>} = \alpha \boldsymbol{S}\boldsymbol{F}^{<t>} + (1-\alpha)\boldsymbol{Y}$
  - ❖ $t = t + 1$
- ❍ 构造未标记样本的预测结果：

$$\hat{y}_i = \arg\max_{j\in\{1,2,\cdots,K\}} (F^*)_{i,j}, i = l+1, l+2, \cdots, N$$

- ❍ 输出结果 $\hat{\boldsymbol{y}} = (\hat{y}_{l+1}, \hat{y}_{l+2}, \cdots, \hat{y}_{l+u})^T$。

9.3 Python 实战

`scikit-learn` 提供了以下两种图半监督学习模型。

- ❑ `LabelPropagation`：它使用标准的迭代式标记传播算法。
- ❑ `LabelSpreading`：它类似`LabelPropagation`，但是使用基于`normalized graph Laplacian and soft clamping`的距离矩阵。

首先导入包：

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import  metrics
from sklearn import datasets
from sklearn.semi_supervised import LabelPropagation
```

然后给出加载数据集的函数`load_data`：

```
def load_data():
    '''
    加载数据集

    :return: 一个元组，依次为： 样本集合、样本标记集合、 未标记样本的下标集合
    '''
    digits = datasets.load_digits()
    ######   混洗样本  ########
    rng = np.random.RandomState(0)
    indices = np.arange(len(digits.data)) # 样本下标集合
    rng.shuffle(indices) # 混洗样本下标集合
    X = digits.data[indices]
    y = digits.target[indices]
    ###### 生成未标记样本的下标集合 ####
    n_labeled_points = int(len(y)/10) # 只有 10% 的样本有标记
    unlabeled_indices = np.arange(len(y))[n_labeled_points:] # 后面 90% 的样本未标记
```

```
return X,y,unlabeled_indices
```

这里使用scikit-learn自带的digits数据集。使用 10% 的样本作为有标记样本，剩余的 90% 样本作为未标记样本。我们并没有直接在原始数据集中修改标记信息，而是返回未标记样本的下标，这是因为后面需要使用原始的标记信息来评判半监督学习的效果。

LabelPropagation

LabelPropagation的原型为：

```
class sklearn.semi_supervised.LabelPropagation(kernel='rbf', gamma=20,
    n_neighbors=7, alpha=1, max_iter=30, tol=0.001)
```

参数

- kernel：一个字符串，指定核函数（也是距离函数）。可以为下列的值。
  - 'rbf'：距离为 $\exp(-\gamma|x-y|^2), \gamma > 0$。
  - 'knn'：如果 x 是 y 的 k 近邻，则距离为 1；否则距离为 0。

> rbf核计算量较大，且距离矩阵是对称的；knn核计算量较小，距离矩阵是稀疏矩阵，且距离矩阵是不对称的。

- gamma：一个浮点数，为 rbf核的参数。
- n_neighbors：一个整数，为knn核的参数。
- alpha：一个浮点数，为折中系数 α。
- max_iter：一个整数，指定最大的迭代次数。
- tol：一个浮点数，指定收敛的阈值。

属性

- X_：输入数组。
- classes_：分类问题中，类别标记数组（每个标记出现一次）。
- label_distributions_：一个数组，给出了每个样本的标记分布。
- transduction_：给出每个样本计算出的标记。
- n_iter_：给出迭代次数。

方法

- fit(X, y)：训练模型。
- predict(X)：预测标记。
- predict_proba(X)：预测对于每个标记出现的概率。
- score(X, y[, sample_weight])：评估在测试集上的预测准确率。

首先给出使用LabelPropagation的函数：

```
def test_LabelPropagation(*data):
    '''
    测试 LabelPropagation 的用法

    :param data: 一个元组，依次为： 样本集合、样本标记集合、 未标记样本的下标集合
    :return: None
    '''
    X,y,unlabeled_indices=data
    y_train=np.copy(y) # 必须复制，后面要用到 y
    y_train[unlabeled_indices]=-1 # 未标记样本的标记设定为 -1
    clf=LabelPropagation(max_iter=100,kernel='rbf',gamma=0.1)
    clf.fit(X,y_train)
    ### 获取预测准确率
    true_labels = y[unlabeled_indices] # 真实标记
    print("Accuracy:%f"%clf.score(X[unlabeled_indices],true_labels))
```

注意：必须调用y_train=np.copy(y)，这是因为后面操作会修改y_train。如果没有这一步复制操作，则会直接修改原始的标记y。而我们需要原始的标记y来评价图半监督学习的效果。另外在这里将未标记样本的标记设置为 -1，表示未标记。

调用test_LabelPropagation函数：

```
data=load_data() # 获取半监督分类数据集
test_LabelPropagation(*data) # 调用 test_LabelPropagation
```

结果如下，可以看到算法对这些未标记样本的预测准确率为 95.8591%。

```
Accuracy:0.958591
```

然后考察折中系数alpha以及 gamma参数对于rbf核的LabelPropagation的预测性能的影响。给出测试函数：

```
def test_LabelPropagation_rbf(*data):
    '''
    测试 LabelPropagation 的 rbf 核时，预测性能随 alpha 和 gamma 的变化

    :param data: 一个元组，依次为： 样本集合、样本标记集合、 未标记样本的下标集合
    :return: None
    '''
    X,y,unlabeled_indices=data
    y_train=np.copy(y) # 必须复制，后面要用到 y
    y_train[unlabeled_indices]=-1 # 未标记样本的标记设定为 -1

    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    alphas=np.linspace(0.01,1,num=10,endpoint=True)
```

```
gammas=np.logspace(-2,2,num=50)
colors=((1,0,0),(0,1,0),(0,0,1),(0.5,0.5,0),(0,0.5,0.5),(0.5,0,0.5),
    (0.4,0.6,0),(0.6,0.4,0),(0,0.6,0.4),(0.5,0.3,0.2),) # 颜色集合，不同的曲线用不同的颜
        色
## 训练并绘图
for alpha,color in zip(alphas,colors):
    scores=[]
    for gamma in gammas:
        clf=LabelPropagation(max_iter=100,gamma=gamma,alpha=alpha,kernel='rbf')
        clf.fit(X,y_train)
        scores.append(clf.score(X[unlabeled_indices],y[unlabeled_indices]))
    ax.plot(gammas,scores,label=r"$\alpha=%s$"%alpha,color=color)

### 设置图形
ax.set_xlabel(r"$\gamma$")
ax.set_ylabel("score")
ax.set_xscale("log")
ax.legend(loc="best")
ax.set_title("LabelPropagation rbf kernel")
plt.show()
```

同样调用test_LabelPropagation_rbf函数，结果如图 9.1 所示。可以看到，当 $\gamma \to +\infty$ 时，距离函数 $\exp(-\gamma|x-y|^2) \to 0$ ，因此当 γ 大到一定程度时，距离矩阵的所有元素都为 0，所以预测准确率会大幅度降低。

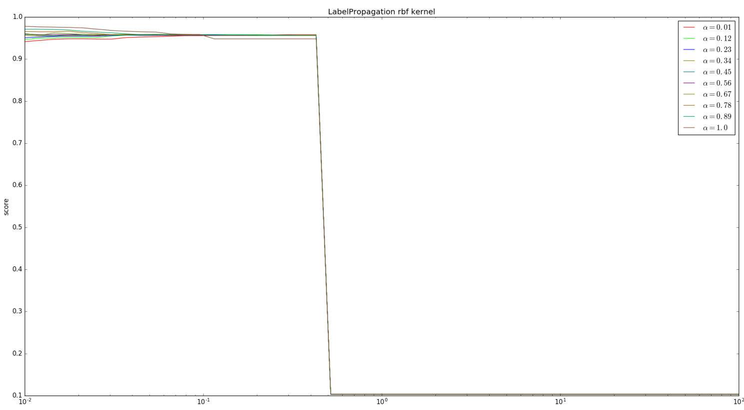


图 9.1 label_propagation_rbf

最后考察折中系数alpha以及 n_neighbors参数对于knn核的LabelPropagation的预测性能的影响。给出测试函数：

```
def test_LabelPropagation_knn(*data):
    '''
   测试 LabelPropagation 的 knn 核时，预测性能随 alpha 和 n_neighbors 的变化
```

```
    :param data:  一个元组，依次为： 样本集合、样本标记集合、 未标记样本的下标集合
    :return:  None
    '''
    X,y,unlabeled_indices=data
    y_train=np.copy(y) # 必须复制，后面要用到 y
    y_train[unlabeled_indices]=-1 # 未标记样本的标记设定为 -1

    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    alphas=np.linspace(0.01,1,num=10,endpoint=True)
    Ks=[1,2,3,4,5,8,10,15,20,25,30,35,40,50]
    colors=((1,0,0),(0,1,0),(0,0,1),(0.5,0.5,0),(0,0.5,0.5),(0.5,0,0.5),
        (0.4,0.6,0),(0.6,0.4,0),(0,0.6,0.4),(0.5,0.3,0.2),) # 颜色集合，不同曲线用不同颜色
    ## 训练并绘图
    for alpha,color in zip(alphas,colors):
        scores=[]
        for K in Ks:
            clf=LabelPropagation(max_iter=100,n_neighbors=K,alpha=alpha,kernel='knn')
            clf.fit(X,y_train)
            scores.append(clf.score(X[unlabeled_indices],y[unlabeled_indices]))
        ax.plot(Ks,scores,label=r"$\alpha=%s$"%alpha,color=color)

    ### 设置图形
    ax.set_xlabel(r"$k$")
    ax.set_ylabel("score")
    ax.legend(loc="best")
    ax.set_title("LabelPropagation knn kernel")
    plt.show()
```

同样调用test_LabelPropagation_knn函数，结果如图 9.2 所示。可以看到：

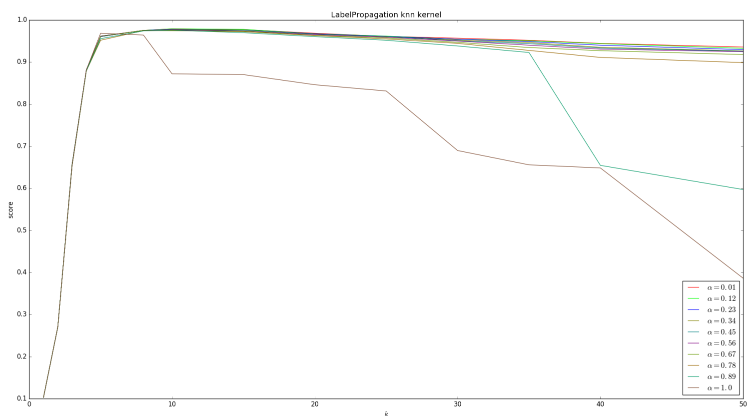


图 9.2 label_propagation_knn

- 随着 k 的增大，预测性能先上升后平稳下降。
- 在 k 较小的时候，α 影响不大。
- 在 k 较大的时候，较小的 α 性能较好。

LabelSpreading

LabelSpreading的原型为：

```
class sklearn.semi_supervised.LabelSpreading(kernel='rbf', gamma=20,
    n_neighbors=7, alpha=0.2, max_iter=30, tol=0.001)
```

参数

- kernel：一个字符串，指定核函数（也是距离函数）。可以为下列的值：
  - 'rbf'：距离为 $\exp(-\gamma|x-y|^2), \gamma > 0$。
  - 'knn'：如果 x 是 y 的 k 近邻，则距离为 1；否则距离为 0。

> rbf核计算量较大，且距离矩阵是对称的；knn核计算量较小，距离矩阵是稀疏矩阵，且距离矩阵是不对称的。

- gamma：一个浮点数，为 rbf核的参数。
- n_neighbors：一个整数，为knn核的参数。
- alpha：一个浮点数，为折中系数 α。
- max_iter：一个整数，指定最大的迭代次数。
- tol：一个浮点数，指定收敛的阈值。

属性

- X_ ：输入数组。
- classes_：分类问题中，类别标记数组（每个标记出现一次）。
- label_distributions_ ：一个数组，给出了每个样本的标记分布。
- transduction_：给出每个样本计算出的标记。
- n_iter_：给出迭代次数。

方法

- fit(X, y)：训练模型。
- predict(X)：预测标记。
- predict_proba(X)：预测对于每个标记出现的概率。
- score(X, y[, sample_weight])：评估在测试集上的预测准确率。

首先给出使用LabelSpreading的函数：

```
def test_LabelSpreading(*data):
    '''
    测试 LabelSpreading 的用法

    :param data: 一个元组，依次为： 样本集合、样本标记集合、 未标记样本的下标集合
    :return: None
    '''
    X,y,unlabeled_indices=data
    y_train=np.copy(y) # 必须复制，后面要用到 y
    y_train[unlabeled_indices]=-1 # 未标记样本的标记设定为 -1
    clf=LabelSpreading(max_iter=100,kernel='rbf',gamma=0.1)
    clf.fit(X,y_train)
    ### 获取预测准确率
    predicted_labels = clf.transduction_[unlabeled_indices] # 预测标记
    true_labels = y[unlabeled_indices] # 真实标记
    print("Accuracy:%f"%metrics.accuracy_score(true_labels,predicted_labels))
```

注意：必须调用y_train=np.copy(y)，这是因为后面的操作会修改y_train。如果没有这一步复制操作，则会直接修改原始的标记y。而我们需要原始的标记y来评价图半监督学习的效果。在这里将未标记样本的标记设置为 −1，表示未标记。另外这里获取预测准确率时，首先预测未标记样本的标记，然后将它与真实标记比较。

调用test_LabelSpreading函数：

```
data=load_data() # 获取半监督分类数据集
test_LabelSpreading(*data) # 调用 test_LabelSpreading
```

结果如下，可以看到算法对这些未标记样本的预测准确率为 97.2806%。

```
Accuracy:0.972806
```

然后考察折中系数alpha以及 gamma参数对于rbf核的LabelSpreading的预测性能的影响。给出测试函数：

```
def test_LabelSpreading_rbf(*data):
    '''
    测试 LabelSpreading 的 rbf 核时，预测性能随 alpha 和 gamma 的变化

    :param data: 一个元组，依次为： 样本集合、样本标记集合、 未标记样本的下标集合
    :return: None
    '''
    X,y,unlabeled_indices=data
    y_train=np.copy(y) # 必须复制，后面要用到 y
    y_train[unlabeled_indices]=-1 # 未标记样本的标记设定为 -1

    fig=plt.figure()
```

```
ax=fig.add_subplot(1,1,1)
alphas=np.linspace(0.01,1,num=10,endpoint=True)
gammas=np.logspace(-2,2,num=50)
colors=((1,0,0),(0,1,0),(0,0,1),(0.5,0.5,0),(0,0.5,0.5),(0.5,0,0.5),
    (0.4,0.6,0),(0.6,0.4,0),(0,0.6,0.4),(0.5,0.3,0.2),) # 颜色集合，不同曲线用不同颜色
## 训练并绘图
for alpha,color in zip(alphas,colors):
    scores=[]
    for gamma in gammas:
        clf=LabelSpreading(max_iter=100,gamma=gamma,alpha=alpha,kernel='rbf')
        clf.fit(X,y_train)
        scores.append(clf.score(X[unlabeled_indices],y[unlabeled_indices]))
    ax.plot(gammas,scores,label=r"$\alpha=%s$"%alpha,color=color)

### 设置图形
ax.set_xlabel(r"$\gamma$")
ax.set_ylabel("score")
ax.set_xscale("log")
ax.legend(loc="best")
ax.set_title("LabelSpreading rbf kernel")
plt.show()
```

同样调用test_LabelSpreading_rbf函数，结果如图 9.3 所示。可以看到，当 $\gamma \to +\infty$ 时，距离函数 $\exp(-\gamma|x-y|^2) \to 0$，因此当 γ 大到一定程度时，距离矩阵的所有元素都为 0，所以预测准确率会大幅度降低。

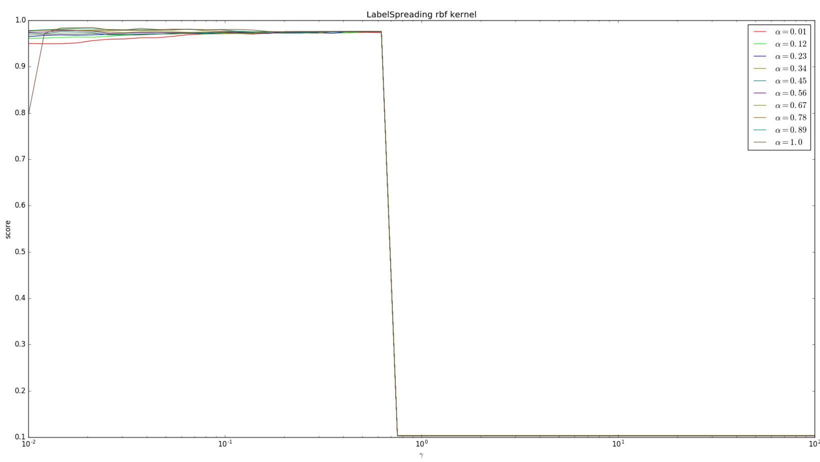


图 9.3　label_spreading_rbf

最后考察折中系数alpha以及 n_neighbors参数对于knn核的LabelSpreading的预测性能的影响。给出测试函数：

```
def test_LabelSpreading_knn(*data):
    '''
   测试 LabelSpreading 的 knn 核时，预测性能随 alpha 和 n_neighbors 的变化

    :param data:  一个元组，依次为： 样本集合、样本标记集合、 未标记样本的下标集合
    :return:  None
    '''
    X,y,unlabeled_indices=data
    y_train=np.copy(y) # 必须复制，后面要用到 y
    y_train[unlabeled_indices]=-1 # 未标记样本的标记设定为 -1

    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    alphas=np.linspace(0.01,1,num=10,endpoint=True)
    Ks=[1,2,3,4,5,8,10,15,20,25,30,35,40,50]
    colors=((1,0,0),(0,1,0),(0,0,1),(0.5,0.5,0),(0,0.5,0.5),(0.5,0,0.5),
        (0.4,0.6,0),(0.6,0.4,0),(0,0.6,0.4),(0.5,0.3,0.2),) # 颜色集合，不同的曲线用不同的颜
            色
    ## 训练并绘图
    for alpha,color in zip(alphas,colors):
        scores=[]
        for K in Ks:
            clf=LabelSpreading(kernel='knn',max_iter=100,n_neighbors=K,alpha=alpha)
            clf.fit(X,y_train)
            scores.append(clf.score(X[unlabeled_indices],y[unlabeled_indices]))
        ax.plot(Ks,scores,label=r"$\alpha=%s$"%alpha,color=color)

    ### 设置图形
    ax.set_xlabel(r"$k$")
    ax.set_ylabel("score")
    ax.legend(loc="best")
    ax.set_title("LabelSpreading knn kernel")
    plt.show()
```

同样调用test_LabelSpreading_knn函数，结果如图 9.4 所示。可以看到：

- 随着 k 的增大，预测性能先上升后平稳下降。
- 在 k 较小的时候，α 影响不大。
- 在 k 较大的时候，较小的 α 性能较好。

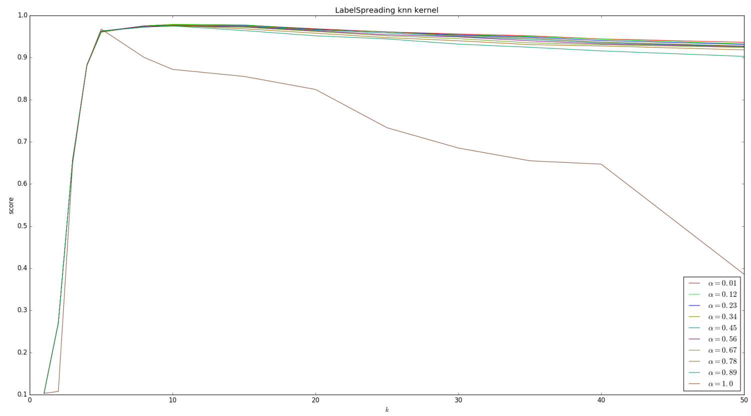


图 9.4　label_spreading_knn

9.4　小结

半监督学习在利用未标记样本后并非必然提升泛化性能，在有些情况下甚至会导致性能下降。对生成式方法，原因通常是模型假设不准确。因此需要依赖充分可靠的领域知识来设计模型。更一般的安全半监督学习仍然是未解决的难题。安全是指：利用未标记样本后，能确保返回性能至少不差于仅利用有标记样本。

第 10 章

集成学习

10.1 概述

集成学习（ensemble learning）是机器学习算法中非常强大的工具，有人把它称为机器学习中的“屠龙刀”，非常万能且有效，在各大机器学习、数据挖掘竞赛中使用非常广泛。

什么是集成学习呢？通俗地讲，就是多算法融合。它的思想相当简单直接，以至于用一句俗语就可以完美概括：三个臭皮匠，顶个诸葛亮。实际操作中，集成学习把大大小小的多种算法融合在一起，共同协作来解决一个问题。这些算法可以是不同的算法，也可以是相同的算法。

集成学习是通过构建并结合多个学习器来完成学习任务。其工作流程为：

- 先产生一组“个体学习器”（individual learner)。在分类问题中，个体学习器也称为基类分类器；
- 再使用某种策略将它们结合起来。

通常使用一种或者多种已有的学习算法从训练数据中产生个体学习器。

10.2 算法笔记精华

10.2.1 集成学习的原理及误差

集成学习通过组合多个个体学习器来获取比单个个体学习器显著优越的泛化性能。通常选取个体学习器的准则是：

- 个体学习器要有一定的准确性，预测能力不能太差；

- ❑ 个体学习器之间要有多样性，即学习器之间要有差异。

> 通常基于实际考虑，人们往往使用预测能力较强的个体学习器（即强学习器，与之对应的为弱学习器）。强学习器的一个显著好处就是可以使用较少数量的个体学习器来集成就可以获得很好的效果。

考虑一个二类分类问题。假设真实类别的取值空间为 $\mathcal{Y}=\{-1,+1\}$。假定基类分类器的错误率为 ϵ，即对每个基分类器 h_i 有：

$$P(h_i(\vec{\boldsymbol{x}}) \neq y)=\epsilon$$

其中 y 为 $\vec{\boldsymbol{x}}$ 的真实类别标记。

假设集成学习结合了 M 个基分类器 $h_1,h_2,\cdots,h_M$。然后通过简单投票法来组合这些基分类器：即若有超过半数的基分类器正确，则集成分类就正确。根据描述，给出集成学习器如下：

$$H(\vec{\boldsymbol{x}})=\operatorname{sign}\left(\sum_{i=1}^{M} h_i(\vec{\boldsymbol{x}})\right)$$

集成学习器预测错误的条件为：k 个基分类器预测正确，其中 $k \leqslant \lfloor M/2 \rfloor$（即少于一半的基分类器预测正确），$M-k$ 个基分类器预测错误。假设基分类器的错误率相互独立，则集成学习器预测错误的概率为：

$$P(H(\vec{\boldsymbol{x}}) \neq y)=\sum_{k=0}^{\lfloor M/2 \rfloor} C_M^k (1-\epsilon)^k \epsilon^{M-k}$$

根据Hoeffding不等式有：

$$\begin{aligned} P(H(\vec{\boldsymbol{x}}) \neq y) &= \sum_{k=0}^{\lfloor M/2 \rfloor} C_M^k (1-\epsilon)^k \epsilon^{M-k} \\ &\leqslant \exp\left(-\frac{1}{2} M (1-2\epsilon)^2\right) \end{aligned}$$

可以看出，随着 $M \to \infty$，集成学习器预测错误的概率 $P(H(\vec{\boldsymbol{x}}) \neq y) \to 0$。

> 但是这里有一个非常关键的地方：假设基分类器的错误率相互独立。实际上这些基分类器的错误率很难相互独立，因为这些基分类器是为了解决同一个问题训练出来的，而且通常是采用相同的算法从同一个训练集中产生的。

根据个体学习器的生成方式，目前的集成学习方法大概可以分为以下两类。

- Boosting算法：在Boosting算法中，个体学习器之间存在强依赖关系、必须串行生成。
- Bagging算法：在Bagging算法中，个体学习器之间不存在强依赖关系、可同时生成。

10.2.2 Boosting 算法

提升方法 (boosting) 是一种常见的统计学习方法。提升方法的理论基础是：强可学习与弱可学习是等价的。在概率近似正确（Probably Approximately Correct,PAC）学习的框架下：

- 强可学习是一个概念（或一个类别），若存在一个多项式的学习算法能够学习它，并且正确率很高，那么称这个概念是强可学习的；
- 弱可学习是一个概念（或一个类别），若存在一个多项式的学习算法能够学习它，学习的正确率仅比随机猜测略好，那么称这个概念是弱可学习的。

可以证明：强可学习与弱可学习是等价的。

> 即若在学习中发现了“弱学习算法”，则可以通过某些办法将它提升为“强学习算法”。

对于分类问题而言，求一个比较粗糙的分类规则（弱分类器）要比求精确的分类规则（强分类器）要容易得多。Boosting 就是一族可以将弱学习器提升为强学习器的算法。其工作原理类似，工作步骤如下：

- 先从初始训练集训练出一个基学习器；
- 再根据基学习器的表现对训练样本权重进行调整，使得被先前的基学习器误判的训练样本在后续受到更多关注；
- 然后基于调整后的样本权重来训练下一个基学习器；
- 如此重复，直到基学习器数量达到给定的值 M 为止；
- 最终将这 M 个基学习器进行加权组合得到集成学习器。

10.2.3 AdaBoost 算法

AdaBoost 的原理

Boosting族算法最常用的是AdaBoost算法。AdaBoot算法包含以下两个核心步骤。

- 权值调整：AdaBoost算法提高那些被前一轮基分类器错误分类样本的权值，而降低那些被正确分类样本的权值。从而使得那些没有得到正确分类的样本，由于权值的加大而受到后一轮基分类器的更大关注。

- 基分类器组合。AdaBoost 采用加权多数表决的方法：
  - 加大分类误差率较小的弱分类器的权值，使得它在表决中起较大的作用；
  - 减小分类误差率较大的弱分类器的权值，使得它在表决中起较小的作用。

AdaBoost 算法

AdaBoost算法如下。

- 输入：
  - 训练数据集 $T=\{(\vec{\boldsymbol{x}}_1,y_1),(\vec{\boldsymbol{x}}_2,y_2),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\}$, $\quad \vec{\boldsymbol{x}}_i\in\mathcal{X}\subset\mathbb{R}^n, y_i\in\mathcal{Y}=\{-1,+1\}$；
  - 弱学习算法。
- 输出：集成分类器 $H(\vec{\boldsymbol{x}})$。
- 算法步骤
  - 初始化训练数据的权重向量 $\vec{\boldsymbol{w}}^{<1>}=(w_1^{<1>},w_2^{<1>},\cdots,w_N^{<1>})^T, w_i^{<1>}=\frac{1}{N}, \quad i=1,2,\cdots,N$。
  - 对 $m=1,2,\cdots,M$
    - 使用具有权重向量 $\vec{\boldsymbol{w}}^{<m>}$ 的训练数据集学习，得到基分类器（根据输入的弱学习算法）：

$$h_m(\vec{\boldsymbol{x}}):\mathcal{X}\to\{-1,+1\}$$

    - 计算 $h_m(\vec{\boldsymbol{x}})$ 在训练数据集上的分类误差率：

$$e_m=P(h_m(\vec{\boldsymbol{x}}_i)\neq y_i)=\sum_{i=1}^N w_i^{<m>}I(h_m(\vec{\boldsymbol{x}}_i)\neq y_i)$$

> 即：所有误分类点的权重之和。权重越大的误差分类点，其在误差率中占比越大。

★ 若 $e_m\geqslant\frac{1}{2}$，算法终止，构建失败！原因见后面的算法解释。

    - 计算 $h_m(\vec{\boldsymbol{x}})$ 的系数（自然对数）：

$$\alpha_m=\frac{1}{2}\log\frac{1-e_m}{e_m}$$

> 该系数表示 $h_m(\vec{\boldsymbol{x}})$ 在最终分类器中的重要性。它是 e_m 的单调减函数（说明误差越小的基本分类器，其重要性越高）。系数大于零也要求 $e_m<\frac{1}{2}$。

- ❖ 更新训练数据集的权重向量：$\vec{\boldsymbol{w}}^{<m+1>} = (w_1^{<m+1>}, w_2^{<m+1>}, \cdots, w_N^{<m+1>})^T$，其中：

$$w_i^{<m+1>} = \frac{w_i^{<m>}}{Z_m} \exp(-\alpha_m y_i h_m(\vec{\boldsymbol{x}}_i)), i = 1, 2, \cdots, N$$

其中 $Z_m = \sum_{i=1}^N w_i^{<m>} \exp(-\alpha_m y_i h_m(\vec{\boldsymbol{x}}_i))$ 为规范化因子。它使得 $\vec{\boldsymbol{w}}^{<m+1>}$ 成为一个概率分布。

> 对于正确分类样本，$h_m(\vec{\boldsymbol{x}}_i) = y_i \rightarrow y_i h_m(\vec{\boldsymbol{x}}_i) = 1$，因此下一轮权重为：$w_i^{<m+1>} = \frac{w_i^{<m>}}{Z_m} \exp(-\alpha_m)$。
> 对于错误分类样本，$h_m(\vec{\boldsymbol{x}}_i) \neq y_i \rightarrow y_i h_m(\vec{\boldsymbol{x}}_i) = -1$，下一轮权重为：$w_i^{<m+1>} = \frac{w_i^{<m>}}{Z_m} \exp(\alpha_m)$。
> 两者比较，误分类样本的权重是正确分类样本的权重的 $\exp(2\alpha_m) = \frac{1-e_m}{e_m}$ 倍。于是误分类样本在下一轮学习中权重更大。

- ❍ 构建基本分类器的线性组合：$f(\vec{\boldsymbol{x}}) = \sum_{m=1}^M \alpha_m h_m(\vec{\boldsymbol{x}})$，于是得到最终分类器：

$$H(\vec{\boldsymbol{x}}) = \text{sign}\left(\sum_{m=1}^M \alpha_m h_m(\vec{\boldsymbol{x}})\right)$$

> 这里实现了 M 个基本分类器的加权表决。系数 α_m 表示基本分类器 $h_m(\vec{\boldsymbol{x}})$ 的重要性，它是 e_m 的单调减函数（说明误差越小的基本分类器，其重要性越高）。

AdaBoost 算法解释

AdaBoost 提高那些被前一轮弱分类器错误分类样本的权值，而降低那些被正确分类样本的权值。这是通过更新训练数据集的权重向量：$\vec{\boldsymbol{w}}^{<m+1>} = (w_1^{<m+1>}, w_2^{<m+1>}, \cdots, w_N^{<m+1>})^T$ 来体现的，其中：

$$w_i^{<m+1>} = \frac{w_i^{<m>}}{Z_m} \exp(-\alpha_m y_i h_m(\vec{\boldsymbol{x}}_i)), i = 1, 2, \cdots, N$$

- ❑ 对于正确分类样本，$h_m(\vec{\boldsymbol{x}}_i) = y_i$ 下一轮权重为：$w_i^{<m+1>} = \frac{w_i^{<m>}}{Z_m} \exp(-\alpha_m)$。
- ❑ 对于错误分类样本，$h_m(\vec{\boldsymbol{x}}_i) \neq y_i$ 下一轮权重为：$w_i^{<m+1>} = \frac{w_i^{<m>}}{Z_m} \exp(\alpha_m)$。

两者比较，误分类样本的权重是正确分类样本权重的 $\exp(2\alpha_m) = \frac{e_m}{1-e_m}$ 倍。考虑到要提高那些被前一轮弱分类器错误分类样本的权值，而降低那些被正确分类样本的权值，就要有 $\exp(2\alpha_m) = \frac{1-e_m}{e_m} > 1$，因此 $e_m < \frac{1}{2}$。因此如果在第 m 次迭代中，发现基本分类器的误差率 $e_m \geqslant \frac{1}{2}$ 时，算法终止。因为不满足算法的条件：提高那些被前一轮弱分类器错误分类样本的权值，而降低那些被正确分类样本的权值。

AdaBoost 采用加权多数表决的方法。加大分类误差率较小的弱分类器的权值，使得它在表决中起较大作用；减小分类误差率较大的弱分类器的权值，使得它在表决中起较小的作用。这是通过 $h_m(\vec{\boldsymbol{x}})$ 的系数（这里对数是自然对数）：

$$\alpha_m = \frac{1}{2}\log\frac{1-e_m}{e_m}$$

来体现的。其中

$$e_m = P(h_m(\vec{\boldsymbol{x}}_i) \neq y_i) = \sum_{i=1}^{N} w_i^{<m>} I(h_m(\vec{\boldsymbol{x}}_i) \neq y_i)$$

AdaBoost 算法误差分析

- 定理一，AdaBoost的训练误差界。AdaBoost算法最终分类器的训练误差上界为：

$$\frac{1}{N}\sum_{i=1}^{N} I(H(\vec{\boldsymbol{x}}_i) \neq y_i) \leq \frac{1}{N}\sum_{i=1}^{N}\exp(-y_i f(\vec{\boldsymbol{x}}_i)) = \prod_{m=1}^{M} Z_m$$

  其中 $Z_m = \sum_{i=1}^{N} w_i^{<m>} \exp(-\alpha_m y_i h_m(\vec{\boldsymbol{x}}_i))$。
- 定理二，二类分类 AdaBoost 的训练误差界：

$$\prod_{m=1}^{M} Z_m = \prod_{m=1}^{M}\left[2\sqrt{e_m(1-e_m)}\,\right] = \prod_{m=1}^{M}\sqrt{(1-4\gamma_m^2)} \leqslant \exp(-2\sum_{m=1}^{M}\gamma_m^2)$$

  其中 $\gamma_m = \frac{1}{2} - e_m$。
- 推论：若存在 $\gamma > 0$，对所有 m，有 $\gamma_m \geqslant \gamma$，则有：

$$\frac{1}{N}\sum_{i=1}^{N} I(H(\vec{\boldsymbol{x}}_i) \neq y_i) \leqslant \exp(-2M\gamma^2)$$

AdaBoost 算法具有自适应性，即它能够适应弱分类器各自的训练误差率。这也是它名字（适应的提升）的由来。从偏差-方差分解的角度来看，AdaBoost主要关注降低偏差，因此AdaBoost能基于弱学习器构建出很强的集成学习器。

AdaBoost 多类分类

标准的AdaBoost算法只适用于二类分类问题，但可以将它推广到多类分类问题。

AdaBoost多类分类算法（SAMME算法）如下。

❑ 输入：
  ❍ 训练数据集 $T=\{(\vec{\boldsymbol{x}}_1,y_1),(\vec{\boldsymbol{x}}_2,y_2),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\}$，$\vec{\boldsymbol{x}}_i\in\mathcal{X}\subset\mathbb{R}^n, y_i\in\mathcal{Y}=\{c_1,c_2,\cdots,c_K\}$；
  ❍ 弱学习算法。

❑ 输出：集成分类器 $H(\vec{\boldsymbol{x}})$。

❑ 算法步骤：
  ❍ 初始化训练数据的权重向量 $\vec{\boldsymbol{w}}^{<1>}=(w_1^{<1>},w_2^{<1>},\cdots,w_N^{<1>})^T, w_i^{<1>}=\frac{1}{N}$，$i=1,2,\cdots,N$。
  ❍ 对 $m=1,2,\cdots,M$
    ❖ 使用具有权重向量 $\vec{\boldsymbol{w}}^{<m>}$ 的训练数据集学习，得到基本分类器（根据输入的弱学习算法）：

$$h_m(\vec{\boldsymbol{x}}):\mathcal{X}\to\{c_1,c_2,\cdots,c_K\}$$

    ❖ 计算 $h_m(\vec{\boldsymbol{x}})$ 在训练数据集上的分类误差率：

$$e_m=P(h_m(\vec{\boldsymbol{x}}_i)\neq y_i)=\sum_{i=1}^{N}w_i^{<m>}I(h_m(\vec{\boldsymbol{x}}_i)\neq y_i)$$

    ❖ 计算 $h_m(\vec{\boldsymbol{x}})$ 的系数（其中对数是自然对数）：

$$\alpha_m=\frac{1}{2}\log\frac{1-e_m}{e_m}+\log(K-1)$$

> 为了系数大于零，要求 $e_m<\frac{K-1}{K}$。

    ❖ 更新训练数据集的权重向量：$\vec{\boldsymbol{w}}^{<m+1>}=(w_1^{<m+1>},w_2^{<m+1>},\cdots,w_N^{<m+1>})^T$，其中

$$w_i^{<m+1>}=\frac{w_i^{<m>}}{Z_m}\exp(-\alpha_m y_i h_m(\vec{\boldsymbol{x}}_i)), i=1,2,\cdots,N$$

    其中 $Z_m=\sum_{i=1}^{N}w_i^{<m>}$，$\exp(-\alpha_m y_i h_m(\vec{\boldsymbol{x}}_i))$ 为规范化因子。它使得 $\vec{\boldsymbol{w}}^{<m+1>}$ 成为一个概率分布。

  ❍ 构建基本分类器的线性组合：$f(\vec{\boldsymbol{x}})=\sum_{m=1}^{M}\alpha_m h_m(\vec{\boldsymbol{x}})$，于是得到最终分类器

$$H(\vec{\boldsymbol{x}})=\arg\max_{c_k}\left(\sum_{m=1}^{M}\alpha_m I(h_m(\vec{\boldsymbol{x}})=c_k)\right)$$

> 即由每个个体分类器带权重投票，得分最多的分类就是集成分类器的分类结果。

当 $K=2$ 时，SAMME算法退化为标准的AdaBoost算法。

AdaBoost还有一种多类分类算法（SAMME.R算法）。该算法需要使用个体分类器对类别进行鉴别的概率。

- 输入：
  - 训练数据集 $T=\{(\vec{\boldsymbol{x}}_1,y_1),(\vec{\boldsymbol{x}}_2,y_2),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\}$，$\vec{\boldsymbol{x}}_i\in\mathcal{X}\subset\mathbb{R}^n, y_i\in\mathcal{Y}=\{c_1,c_2,\cdots,c_K\}$。
  - 弱学习算法。
- 输出：集成分类器 $H(\vec{\boldsymbol{x}})$。
- 算法步骤：
  - 初始化训练数据的权重向量 $\vec{\boldsymbol{w}}^{<1>}=(w_1^{<1>},w_2^{<1>},\cdots,w_N^{<1>})^T, w_i^{<1>}=\frac{1}{N},\quad i=1,2,\cdots,N$。
  - 对 $m=1,2,\cdots,M$
    - 使用具有权重向量 $\vec{\boldsymbol{w}}^{<m>}$ 的训练数据集学习，得到基本分类器（根据输入的弱学习算法）：

$$h_m(\vec{\boldsymbol{x}}):\mathcal{X}\to\{c_1,c_2,\cdots,c_K\}$$

    - 计算 $h_m(\vec{\boldsymbol{x}})$ 在训练数据集上的加权概率估计：

$$p_{m,i}^{(k)}=w_i^{<m>}P(y_i=c_k/\vec{\boldsymbol{x}}_i), i=1,2,\cdots,N; k=1,2,\cdots,K$$

    其中 y_i 是 $\vec{\boldsymbol{x}}_i$ 的真实标记，$w_i^{<m>}$ 是 $\vec{\boldsymbol{x}}_i$ 的权重。$p_{m,i}^{(k)}$ 刻画了对于 h_m，它预测 $\vec{\boldsymbol{x}}_i$ 的输出为每种类别的概率的加权值。

    - 对 h_m 和类别 c_k，定义：

$$l_m^{(k)}(\vec{\boldsymbol{x}}_i)=(K-1)\left(\log p_{m,i}^{(k)}-\frac{1}{K}\sum_{k=1}^{K}\log p_{m,i}^{(k)}\right), k=1,2,\cdots,K$$

    - 更新训练数据集的权重向量 $\vec{\boldsymbol{w}}^{<m+1>}=(w_1^{<m+1>},w_2^{<m+1>},\cdots,w_N^{<m+1>})^T$：

$$w_i^{<m+1>}=w_i^{<m>}\exp\left(-\frac{K-1}{K}\sum_{k=1}^{K}\delta_i^{(k)}\log p_{m,i}^{(k)}\right)$$

    其中

$$\delta_i^{(k)}=\begin{cases}1, & \text{if}\quad y_i=c_k\\ -\frac{1}{K-1}, & \text{else}\end{cases}$$

- 归一化训练数据集的权重向量 $\vec{\boldsymbol{w}}^{<m+1>} = (w_1^{<m+1>}, w_2^{<m+1>}, \cdots, w_N^{<m+1>})^T$，使得权值之和为 1。
- 构建基本分类器的线性组合，得到最终分类器：

$$H(\vec{\boldsymbol{x}}) = \arg\max_{c_k} \left(\sum_{m=1}^{M} l_m^{(k)}(\vec{\boldsymbol{x}}) \right)$$

10.2.4 AdaBoost 与加法模型

AdaBoost 算法可以认为是：模型为加法模型，损失函数为指数函数，学习算法为前向分步算法时的二类分类学习方法。

首先给出加法模型：$f(\vec{\boldsymbol{x}}) = \sum_{m=1}^{M} \beta_m b(\vec{\boldsymbol{x}}; \gamma_m)$，其中 $b(\vec{\boldsymbol{x}}; \gamma_m)$ 为基函数，γ_m 为基函数的参数，β_m 为基函数的系数。

给定训练数据以及损失函数 $L(y, f(\vec{\boldsymbol{x}}))$，求解加法模型的目标是：损失函数极小化：

$$\min_{\beta_m, \gamma_m} \sum_{i=1}^{N} L\left(y_i, \sum_{m=1}^{M} \beta_m b(\vec{\boldsymbol{x}}; \gamma_m) \right)$$

解决这个问题的方法是前向分步算法。前向分步算法的思想是：从前向后，每一步只求解一个基函数及其系数，逐步逼近优化目标函数。因此每一步只需要优化如下的损失函数：

$$\min_{\beta, \gamma} \sum_{i=1}^{N} L(y_i, \beta b(\vec{\boldsymbol{x}}; \gamma))$$

这里给出前向分步算法如下。

- 输入：
  - 训练数据集 $T = \{(\vec{\boldsymbol{x}}_1, y_1), (\vec{\boldsymbol{x}}_2, y_2), \cdots, (\vec{\boldsymbol{x}}_N, y_N)\}, \quad \vec{\boldsymbol{x}}_i \in \mathcal{X} \subset \mathbb{R}^n, y_i \in \mathcal{Y} = \{-1, +1\}$。
  - 损失函数 $L(y, f(\vec{\boldsymbol{x}}))$。
  - 基函数集 $\{b(\vec{\boldsymbol{x}}; \gamma)\}$。
- 输出：加法模型 $f(\vec{\boldsymbol{x}})$。
- 算法步骤：
  - 初始化 $f_0(\vec{\boldsymbol{x}}) = 0$。
  - 对 $m = 1, 2, \cdots, M$

❖ 极小化损失函数为

$$(\beta_m,\gamma_m)=\arg\min_{\beta,\gamma}\sum_{i=1}^{N}L(y_i,f_{m-1}(\vec{\boldsymbol{x}}_i)+\beta b(\vec{\boldsymbol{x}}_i;\gamma))$$

得到参数 β_m,γ_m。

❖ 更新 $f_m(\vec{\boldsymbol{x}})=f_{m-1}(\vec{\boldsymbol{x}})+\beta_m b(\vec{\boldsymbol{x}};\gamma_m)$。

❍ 得到加法模型：$f(\vec{\boldsymbol{x}})=f_M(\vec{\boldsymbol{x}})=\sum_{i=1}^{M}\beta_m b(\vec{\boldsymbol{x}};\gamma_m)$。

AdaBoost 算法是前向分步加法算法的特例。此时模型是由基本分类器组成的加法模型，损失函数是指数函数。其中指数损失函数为：

$$L(y,f(\vec{\boldsymbol{x}}))=e^{-yf(\vec{\boldsymbol{x}})}$$

10.2.5 提升树

提升树的原理

提升树（boostring tree）是以决策树为基本学习器的提升方法，其预测性能相当优异。

❑ 对分类问题，决策树是二叉决策树。
❑ 对回归问题，决策树是二叉回归树。

提升树模型可以表示为决策树为基本分类器的加法模型：

$$f(\vec{\boldsymbol{x}})=f_M(\vec{\boldsymbol{x}})=\sum_{m=1}^{M}h_m(\vec{\boldsymbol{x}};\Theta_m)$$

其中 $h_m(\vec{\boldsymbol{x}};\Theta_m)$ 表示决策树；Θ_m 为决策树的参数；M 为决策树的数量。

提升树算法采用前向分步算法。令 $f_{m-1}(\vec{\boldsymbol{x}})$ 为当前模型，则第 m 步模型为：

$$f_m(\vec{\boldsymbol{x}})=f_{m-1}(\vec{\boldsymbol{x}})+h_m(\vec{\boldsymbol{x}};\Theta_m)$$

其中初始提升树 $f_0(\vec{\boldsymbol{x}})=0$。

通过损失函数最小化来确定下一棵决策树的参数 Θ_m：

$$\hat{\Theta}_m=\arg\min_{\Theta_m}\sum_{i=1}^{N}L(y_i,f_m(\vec{\boldsymbol{x}}))$$

若使用不同的损失函数，则得到如下两种不同的提升树学习算法（设预测值为 $\hat{y}$。真实值为 y)：

❑ 回归问题：通常使用损失函数，用平方误差损失函数

$$L(y,\hat{y})=(y-\hat{y})^2$$

❑ 分类问题：通常使用损失函数，用指数损失函数

$$L(y,\hat{y})=e^{-y\hat{y}}$$

提升树算法

给定训练数据集 $T=\{(\vec{\boldsymbol{x}}_1,y_1),(\vec{\boldsymbol{x}}_2,y_2),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\}$， $\vec{\boldsymbol{x}}_i\in\mathcal{X}\subseteq\mathbb{R}^n, y_i\in\mathcal{Y}\subseteq\mathbb{R}$，其中 $\mathcal{X}$ 为输入空间，$\mathcal{Y}$ 为输出空间。

如果将输入空间 $\mathcal{X}$ 划分为 J 个互不相交的区域 $\boldsymbol{R}_1,\boldsymbol{R}_2,\cdots,\boldsymbol{R}_J$，并且在每个区域上确定输出的常量 c_j，则树可以表示为：$h(\vec{\boldsymbol{x}};\Theta)=\sum_{j=1}^J c_j I(\vec{\boldsymbol{x}}\in\boldsymbol{R}_j)$。

其中参数 $\Theta=\{(\boldsymbol{R}_1,c_1),(\boldsymbol{R}_2,c_2),\cdots,(\boldsymbol{R}_J,c_J)\}$ 表示树的划分区域和各区域上的输出。J 是回归树的复杂度，即叶节点个数。

采用前向分步算法，则在第 m 步给定当前模型 $f_{m-1}(\vec{\boldsymbol{x}})$，求解：

$$\hat{\Theta}_m=\arg\min_{\Theta_m}\sum_{i=1}^N L(y_i,f_{m-1}(\vec{\boldsymbol{x}}_i)+h_m(\vec{\boldsymbol{x}}_i;\Theta_m))$$

得到 $\hat{\Theta}_m$，即第 m 棵树的参数。

对于回归问题，如果采用平方误差损失函数 $L(y,f(\vec{\boldsymbol{x}}))=(y-f(\vec{\boldsymbol{x}}))^2$，其损失为：

$$L(y,f_{m-1}(\vec{\boldsymbol{x}})+h_m(\vec{\boldsymbol{x}};\Theta_m))=[y-f_{m-1}(\vec{\boldsymbol{x}})-h_m(\vec{\boldsymbol{x}};\Theta_m)]^2=[r-h_m(\vec{\boldsymbol{x}};\Theta_m)]^2$$

注意： $r=y-f_{m-1}(\vec{\boldsymbol{x}})$ 刚好就是当前模型拟合数据的残差（是利用当前模型 $f_{m-1}(\vec{\boldsymbol{x}})$ 拟合训练数据集得到的）。所以对回归问题的提升树算法，只需要简单地用 $h_m(\vec{\boldsymbol{x}};\Theta_m)$ 拟合当前模型的残差即可。

回归问题的提升树算法如下。

❑ 输入：训练数据集 $T=\{(\vec{\boldsymbol{x}}_1,y_1),(\vec{\boldsymbol{x}}_2,y_2),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\}$， $\vec{\boldsymbol{x}}_i\in\mathcal{X}\subseteq\mathbb{R}^n, y_i\in\mathcal{Y}\subseteq\mathbb{R}$。
❑ 输出：提升树 $f_M(\vec{\boldsymbol{x}})$。
❑ 算法步骤：
  ❍ 初始化 $f_0(\vec{\boldsymbol{x}})=0$
  ❍ 对于 $m=1,2,\cdots,M$
    ❖ 计算残差 $r_{mi}=y_i-f_{m-1}(\vec{\boldsymbol{x}}_i),i=1,2,\cdots,N$。
    ❖ 拟合残差 r_{mi} 学习一棵回归树，得到 $h_m(\vec{\boldsymbol{x}};\Theta_m)$。

> 拟合残差：生成一棵回归树，使得它的损失函数最小。

- ❖ 更新 $f_m(\vec{\boldsymbol{x}}) = f_{m-1}(\vec{\boldsymbol{x}}) + h_m(\vec{\boldsymbol{x}};\Theta_m)$。
- ❍ 得到提升树 $f_M(\vec{\boldsymbol{x}}) = \sum_{m=1}^{M} h_m(\vec{\boldsymbol{x}};\Theta_m)$。

GBDT & GBRT

在提升树中，如果损失函数是平方损失函数和指数损失函数时，由于这两种函数的求导很简单，所以求解：

$$\hat{\Theta}_m = \arg\min_{\Theta_m} \sum_{i=1}^{N} L(y_i, f_{m-1}(\vec{\boldsymbol{x}}_i) + h_m(\vec{\boldsymbol{x}}_i;\Theta_m))$$

这一最优化问题比较简单。当损失函数是一般函数时，该最优化问题往往很难求得。

Freidman提出了梯度提升算法来解决这个问题。利用损失函数的负梯度在当前模型的值作为提升树算法中残差的近似值，拟合一棵决策树。在回归问题中，这称为梯度提升回归树GBRT；在分类问题中，这称为梯度提升决策树GBDT。

梯度提升树算法如下。

- ❑ 输入：
  - ❍ 训练数据集 $T = \{(\vec{\boldsymbol{x}}_1, y_1), (\vec{\boldsymbol{x}}_2, y_2), \cdots, (\vec{\boldsymbol{x}}_N, y_N)\}, \quad \vec{\boldsymbol{x}}_i \in \mathcal{X} \subseteq \mathbb{R}^n, y_i \in \mathcal{Y} \subseteq \mathbb{R}$。

    > 如果为分类问题，则 $\mathcal{Y} = \{-1, +1\}$。

  - ❍ 损失函数 $L(y, f(\vec{\boldsymbol{x}}))$。
- ❑ 输出：回归树 $f_M(\vec{\boldsymbol{x}})$。
- ❑ 算法步骤：
  - ❍ 初始化

$$f_0(\vec{\boldsymbol{x}}) = \arg\min_{c} \sum_{i=1}^{N} L(y_i, c)$$

    > 它是一棵只有根节点的树，根节点的类别为使得损失函数最小的值。

  - ❍ 对于 $m = 1, 2, \cdots, M$

❖ 对于 $i = 1, 2, \cdots, N$, 计算

$$r_{mi} = -\left[\frac{\partial L(y_i, f(\vec{\boldsymbol{x}}_i))}{\partial f(\vec{\boldsymbol{x}}_i)}\right]_{f(\vec{\boldsymbol{x}})=f_{m-1}(\vec{\boldsymbol{x}})}$$

> 计算损失函数的负梯度在当前模型的值，将它作为残差估计。
> 对于平方损失函数，它就是通常意义上的残差。
> 对于一般损失函数，它就是残差的近似。

❖ 对 r_{mi} 拟合一棵回归树，得到第 m 棵树的叶节点区域 $R_{mj}, j = 1, 2, \cdots, J$。
❖ 计算在每个区域 $\boldsymbol{R}_{mj}$ 上的输出值：对 $j = 1, 2, \cdots, J$ 计算：

$$c_{mj} = \arg\min_{c} \sum_{\vec{\boldsymbol{x}}_i \in \boldsymbol{R}_{mj}} L(y_i, f_{m-1}(\vec{\boldsymbol{x}}_i) + c)$$

❖ 更新 $f_m(\vec{\boldsymbol{x}}) = f_{m-1}(\vec{\boldsymbol{x}}) + \sum_{j=1}^{J} c_{mj} I(\vec{\boldsymbol{x}} \in \boldsymbol{R}_{mj})$

> 这一步是计算出了决策树 $h_m(\vec{\boldsymbol{x}}) = \sum_{j=1}^{J} c_{mj} I(\vec{\boldsymbol{x}} \in \boldsymbol{R}_{mj})$，这棵决策树将 $\vec{\boldsymbol{x}}$ 所属的叶节点的输出作为 $\vec{\boldsymbol{x}}$ 的预测值。

❍ 得到回归树：

$$f_M(\vec{\boldsymbol{x}}) = \sum_{m=1}^{M} \sum_{j=1}^{J} c_{mj} I(\vec{\boldsymbol{x}} \in \boldsymbol{R}_{mj})$$

上述梯度提升树算法如果应用于回归问题，就是GBRT；如果应用于分类问题，就是GBDT。

10.2.6 Bagging 算法

Bagging 算法原理

Bagging基于自助采样法（bootstrap sampling）。给定包含 N 个样本的训练数据集 D，自助采样法是这样进行的：先从 D 中随机取出一个样本放入采样集 D_s 中，再把该样本放回 D 中（有放回的重复独立采样）。经过 N 次随机采样操作，得到包含 N 个样本的采样集 D_s。

注意：数据集 D 中可能有的样本在采样集 D_s 中多次出现，但是 D 中也可能有样本在 D_s 中从未出现。一个样本始终不在采样集中出现的概率是 $(1-\frac{1}{N})^N$ 。根据：

$$\lim_{N\to\infty}(1-\frac{1}{N})^N = \frac{1}{e} \simeq= 0.368$$

因此 D 中约有 63.2% 的样本出现在了 D_s 中。

Bagging 首先采用 M 轮自助采样法，获得 M 个包含 N 个训练样本的采样集。然后，基于这些采样集训练出一个基学习器。最后将这 M 个基学习器进行组合。组合策略为：

- 分类任务采取简单投票法，即每个基学习器一票；
- 回归任务使用简单平均法，即每个基学习器的预测值取平均值。

从偏差-方差分解的角度来看，Bagging主要关注降低方差。因此它在不剪枝决策树、神经网络等容易受到样本扰动的学习器上效果更为明显。

随机森林

随机森林（Random Forest，RF）：RF是一种以决策树为基学习器的Bagging算法，但是RF在决策树的训练过程中引入了随机属性选择。

- 传统决策树在选择划分属性时，每次选择一个最优属性。
- 在RF中构建决策树，选择节点的划分属性时：
  - 首先从该节点的属性集合中随机选择一个包含 k 个属性的子集；
  - 然后再从这个子集中选择一个最优属性用于划分。

  如果 $k=n$（其中 n 为当前节点的属性的数量），则RF中决策树的构建与传统决策树相同。如果 $k=1$，则随机选择一个属性用于划分。通常建议 $k=\log_2 n$。

RF的训练效率较高，因为RF使用的决策树只需要考虑一个属性的子集。另外，RF简单、容易实现、计算开销小，而且它在很多现实任务中展现出强大的性能。

10.2.7 误差-分歧分解

假定在回归问题中，有 M 个个体学习器 $h_1,h_2,\cdots,h_M$ ，通过加权平均法组合产生集成学习器 H 。即

$$H(\vec{\boldsymbol{x}})=\sum_{i=1}^{M}w_i h_i(\vec{\boldsymbol{x}})$$

对于某个样本 $\vec{\boldsymbol{x}}$，定义学习器 h_i 的分歧ambiguity为：

$$A(h_i\mid\vec{\boldsymbol{x}})=(h_i(\vec{\boldsymbol{x}})-H(\vec{\boldsymbol{x}}))^2$$

集成学习器的分歧为:

$$\bar{A}(H\mid\vec{\boldsymbol{x}})=\sum_{i=1}^{M}w_i A(h_i\mid\vec{\boldsymbol{x}})=\sum_{i=1}^{M}w_i(h_i(\vec{\boldsymbol{x}})-H(\vec{\boldsymbol{x}}))^2$$

分歧刻画了个体学习器在某个样本 $\vec{\boldsymbol{x}}$ 上的不一致性，在一定程度上反映了个体学习器的多样性。

设样本 $\vec{\boldsymbol{x}}$ 的真实标记为 y，则个体学习器 h_i 和集成学习器 H 的平方误差为：

$$e_i(\vec{\boldsymbol{x}}) = \left(y - h_i(\vec{\boldsymbol{x}})\right)^2 \ e_H(\vec{\boldsymbol{x}}) = \left(y - H(\vec{\boldsymbol{x}})\right)^2$$

令个体学习器误差的加权均值为：

$$\bar{e}_h(\vec{\boldsymbol{x}}) = \sum_{i=1}^{M} w_i e_i(\vec{\boldsymbol{x}})$$

根据 $H(\vec{\boldsymbol{x}}) = \sum_{i=1}^{M} w_i h_i(\vec{\boldsymbol{x}})$ 则有：

$$\bar{A}(H \mid \vec{\boldsymbol{x}}) = \sum_{i=1}^{M} w_i e_i(\vec{\boldsymbol{x}}) - e_H(\vec{\boldsymbol{x}}) = \bar{e}_h(\vec{\boldsymbol{x}}) - e_H(\vec{\boldsymbol{x}})$$

上式对所有样本 $\vec{\boldsymbol{x}}$ 成立。令 $p(\vec{\boldsymbol{x}})$ 为样本的概率密度，则在全样本上有：

$$\int \bar{A}(H \mid \vec{\boldsymbol{x}}) p(\vec{\boldsymbol{x}}) \mathrm{d}\vec{\boldsymbol{x}} = \int \bar{e}_h(\vec{\boldsymbol{x}})\ p(\vec{\boldsymbol{x}}) \mathrm{d}\vec{\boldsymbol{x}} - \int e_H(\vec{\boldsymbol{x}})\ p(\vec{\boldsymbol{x}}) \mathrm{d}\vec{\boldsymbol{x}}$$

代入各变量，则有：

$$\int \sum_{i=1}^{M} w_i A(h_i \mid \vec{\boldsymbol{x}}) p(\vec{\boldsymbol{x}}) \mathrm{d}\vec{\boldsymbol{x}} = \int \sum_{i=1}^{M} w_i e_i(\vec{\boldsymbol{x}})\ p(\vec{\boldsymbol{x}}) \mathrm{d}\vec{\boldsymbol{x}} - \int e_H(\vec{\boldsymbol{x}})\ p(\vec{\boldsymbol{x}}) \mathrm{d}\vec{\boldsymbol{x}}$$
$$\rightarrow \sum_{i=1}^{M} w_i \int A(h_i \mid \vec{\boldsymbol{x}}) p(\vec{\boldsymbol{x}}) \mathrm{d}\vec{\boldsymbol{x}} = \sum_{i=1}^{M} w_i \int e_i(\vec{\boldsymbol{x}})\ p(\vec{\boldsymbol{x}}) \mathrm{d}\vec{\boldsymbol{x}} - \int e_H(\vec{\boldsymbol{x}})\ p(\vec{\boldsymbol{x}}) \mathrm{d}\vec{\boldsymbol{x}}$$

定义个体学习器 h_i 在全体样本上的泛化误差和分歧项为：

$$E_i = \int e_i(\vec{\boldsymbol{x}})\ p(\vec{\boldsymbol{x}}) \mathrm{d}\vec{\boldsymbol{x}}$$
$$A_i = \int A(h_i \mid \vec{\boldsymbol{x}})\ p(\vec{\boldsymbol{x}}) \mathrm{d}\vec{\boldsymbol{x}}$$

定义集成的泛化误差为：

$$E = \int e_H(\vec{\boldsymbol{x}})\ p(\vec{\boldsymbol{x}}) \mathrm{d}\vec{\boldsymbol{x}}$$

则有：

$$\sum_{i=1}^{M} w_i A_i = \sum_{i=1}^{M} w_i E_i - E$$

定义个体学习器泛化误差的加权均值为 $\bar{E} = \sum_{i=1}^{M} w_i E_i$；定义个体学习器的加权分歧值为 $\bar{A} = \sum_{i=1}^{M} w_i A_i$，则有：

$$E = \bar{E} - \bar{A}$$

上式就是集成学习的误差-分歧分解。从中可以看出：要想降低集成学习的泛化误差 E，要么提高个体学习器的加权分歧值 $\bar{A}$，要么降低个体学习器的泛化误差的加权均值 $\bar{E}$。因此：个体学习器准确性越高、多样性越大，则集成越好。

> 该式难以直接作为优化目标，因为现实任务中很难直接对 $\bar{E} - \bar{A}$ 进行优化：
> 一方面它们是定义在整体样本空间上的，而不是训练集上的；
> 另一方面 $\bar{A}$ 是当你集成学习器构造之后才能进行估计的，但是集成学习器此时未知。

10.2.8 多样性增强

集成学习中，个体学习器多样性越大越好。通常为了增大个体学习器的多样性，在学习过程中引入随机性。常见的方法有：对数据样本进行扰动、对输入属性进行扰动、对算法参数进行扰动三种。

数据样本扰动：给定初始数据集，可以使用采样法从中产生出不同的数据子集。然后再利用不同的数据子集训练出不同的个体学习器。这种方法简单高效，使用广泛。

- 数据样本扰动对于“不稳定基学习器”很有效。“不稳定基学习器”是这样的一类学习器：训练样本稍加变化就会导致学习器有显著的变动，如决策树、神经网络等。
- 数据样本扰动对于“稳定基学习器”无效。“稳定基学习器”是这样的一类学习器：学习器对于数据样本的扰动敏感，如线性学习器、支持向量机、朴素贝叶斯、k 近邻学习器等。

输入属性扰动：训练样本通常由一组属性来描述，可以基于这些属性的不同组合产生不同的数据子集，然后再利用这些数据子集训练出不同的个体学习器。

- 若数据包含了大量冗余属性，则输入属性扰动的效果较好。此时不仅训练出了多样性大的个体，还会因为属性数量的减少而大幅节省时间开销。同时由于冗余属性多，即使减少一些属性，训练的个体学习器也不会很差。
- 若数据只包含少量属性，则不宜采用输入属性扰动法。

算法参数扰动：通常可以通过随机设置不同的参数，从而产生差别较大的个体学习器。

> 在使用交叉验证法来确定基学习器的参数时，实际上就是用不同的参数训练出来了多个学习器，然后从中挑选出效果最好的那个学习器。集成学习则相当于把所有这些学习器都利用起来了。

可以同时使用不同的多样性增强机制，如随机森林同时用了数据样本扰动和输入属性扰动。

10.3　Python 实战

首先导入包：

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets,cross_validation,ensemble
```

在回归问题中，使用的数据集是scikit-learn自带的一个糖尿病病人的数据集。该数据集从糖尿病病人采样并整理后，特点如下：

- 数据集有 442 个样本；
- 每个样本有 10 个特征；
- 每个特征都是浮点数，数据都在 −0.2 ~ 0.2 之间；
- 样本的目标在整数 25 ~ 346 之间。

这里给出加载数据集的函数：

```
def load_data_regression():
    diabetes = datasets.load_diabetes()
    return cross_validation.train_test_split(diabetes.data,diabetes.target,
        test_size=0.25,random_state=0)
```

- 返回值：一个元组，元组依次是：训练样本集、测试样本集、训练样本集对应的标签值、测试样本集对应的标签值。

在分类问题中，使用的是scikit-learn自带的手写识别数据集Digit Dataset。该数据集由 1797 张样本图片组成。每张样本图片都是一个 8×8 大小的手写数字位图。

然后给出加载数据集的函数：

```
def load_data_classification():
    digits=datasets.load_digits()
    return cross_validation.train_test_split(digits.data,digits.target,
        test_size=0.25,random_state=0)
```

- 返回值：一个元组，元组依次是：训练样本集、测试样本集、训练样本集对应的标签值、测试样本集对应的标签值。

在这里采用分层采样。分层采样保证了测试样本集中各类别样本的比例与原始样本集中各类别样本的比例相同。如果不采取分层采用，那么最后切分得到的测试数据集就不是无偏的了。

10.3.1 AdaBoost

scikit-learn基于AdaBoost算法提供了两个模型：

- AdaBoostClassifier用于分类问题；
- AdaBoostRegressor用于回归问题。

AdaBoostClassifier 分类器

AdaBoostClassifier是scikit-learn提供的AdaBoost分类器，其原型为：

```
class sklearn.ensemble.AdaBoostClassifier(base_estimator=None, n_estimators=50,
learning_rate=1.0, algorithm='SAMME.R', random_state=None)
```

参数

- base_estimator是一个基础分类器对象。默认为DecisionTreeClassfier。该基础分类器必须支持带样本权重的学习。
- n_estimators是一个整数，指定基础分类器的数量（默认为 50）。当然如果训练集已经完美地训练好了，可能算法会提前停止。此时基础分类器数量少于该值。
- learning_rate为浮点数。默认为 1。它用于减少每一步的步长，防止步长太大而跨过了极值点。通常learning_rate越小，则需要的基础分类器数量会越多，因此在learning_rate和n_estimators之间会有所折中。learning_rate就是下式中的 ν。

$$H_m(\vec{\boldsymbol{x}}) = H_{m-1}(\vec{\boldsymbol{x}}) + \nu\alpha_m h_m(\vec{\boldsymbol{x}})$$

- algorithm为一个字符串，指定算法，该算法用于多类分类问题。默认为'SAMME.R'。
  - 'SAMME.R'：使用SAMME.R 算法。基础分类器对象必须支持计算类别的概率。
  - 'SAMME'：使用SAMME算法。

> 通常'SAMME.R' 收敛更快，且误差更小，迭代数量更少。

- random_state：一个整数，或者一个RandomState实例，或者None。

- 如果为整数，则它指定随机数生成器的种子。
- 如果为RandomState实例，则指定随机数生成器。
- 如果为None，则使用默认的随机数生成器。

属性

- estimators_：所有训练过的基础分类器。
- classes_：所有的类别标签。
- n_classes_：类别数量。
- estimator_weights_：每个基础分类器的权重。
- estimator_errors_：每个基础分类器的分类误差。
- feature_importances_：每个特征的重要性。

方法

- fit(X, y[, sample_weight])：训练模型。
- predict(X)：用模型进行预测，返回预测值。
- predict_log_proba(X)：返回一个数组，数组的元素依次是X预测为各个类别的概率的对数值。
- predict_proba(X)：返回一个数组，数组的元素依次是X预测为各个类别的概率值。
- score(X,y[,sample_weight])：返回在(X,y)上预测的准确率(accuracy)。
- staged_predict(X)：返回一个数组，数组元素依次是每一轮迭代结束时尚未完成的集成分类器的预测值。
- staged_predict_proba(X)：返回一个二维数组，数组元素依次是每一轮迭代结束时尚未完成的集成分类器预测X为各个类别的概率值。
- staged_score(X, y[, sample_weight])：返回一个数组，数组元素依次是每一轮迭代结束时尚未完成的集成分类器的预测准确率。

首先给出使用AdaBoostClassifier的函数：

```
def test_AdaBoostClassifier(*data):
    X_train,X_test,y_train,y_test=data
    clf=ensemble.AdaBoostClassifier(learning_rate=0.1)
    clf.fit(X_train,y_train)
    ## 绘图
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    estimators_num=len(clf.estimators_)
    X=range(1,estimators_num+1)
    ax.plot(list(X),list(clf.staged_score(X_train,y_train)),label="Traing score")
    ax.plot(list(X),list(clf.staged_score(X_test,y_test)),label="Testing score")
    ax.set_xlabel("estimator num")
    ax.set_ylabel("score")
```

```
    ax.legend(loc="best")
    ax.set_title("AdaBoostClassifier")
    plt.show()
```

原本打算都使用默认值，但是默认的learning_rate=1太大导致效果不佳，调用该函数

```
X_train,X_test,y_train,y_test=load_data_classification()
test_AdaBoostClassifier(X_train,X_test,y_train,y_test)
```

运行结果如图 10.1 所示。可以看到随着算法的推进，每一轮迭代都产生一个新的个体分类器被集成。此时的集成分类器的训练误差和测试误差都在下降（对应的就是训练准确率和测试准确率上升）。当个体分类器数量达到一定值时，集成分类器的预测准确率在一定范围内波动，比较稳定。这也证实了前文的论述：集成学习能很好地抵抗过拟合。

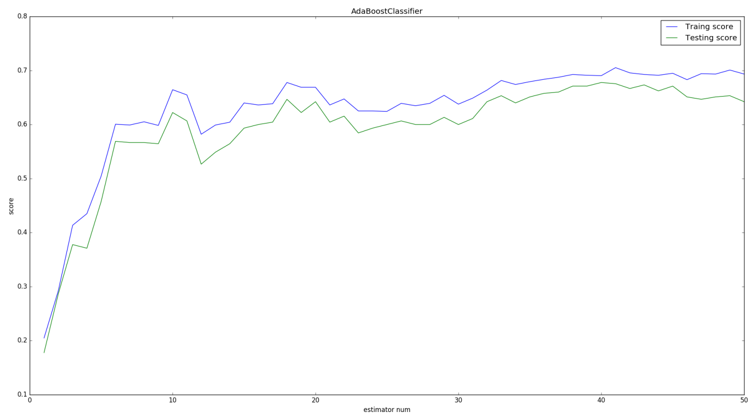


图 10.1　AdaBoostClassifier

下面考察不同类型的个体分类器的影响，给出测试函数：

```
def test_AdaBoostClassifier_base_classifier(*data):
    from sklearn.naive_bayes import  GaussianNB
    X_train,X_test,y_train,y_test=data
    fig=plt.figure()
    ax=fig.add_subplot(2,1,1)

    ########### 默认的个体分类器  #############
    clf=ensemble.AdaBoostClassifier(learning_rate=0.1)
    clf.fit(X_train,y_train)
    ## 绘图
    estimators_num=len(clf.estimators_)
    X=range(1,estimators_num+1)
    ax.plot(list(X),list(clf.staged_score(X_train,y_train)),label="Traing score")
    ax.plot(list(X),list(clf.staged_score(X_test,y_test)),label="Testing score")
    ax.set_xlabel("estimator num")
    ax.set_ylabel("score")
    ax.legend(loc="lower right")
```

```
ax.set_ylim(0,1)
ax.set_title("AdaBoostClassifier with Decision Tree")
####### Gaussian Naive Bayes 个体分类器  ########
ax=fig.add_subplot(2,1,2)
clf=ensemble.AdaBoostClassifier(learning_rate=0.1,base_estimator=GaussianNB())
clf.fit(X_train,y_train)
## 绘图
estimators_num=len(clf.estimators_)
X=range(1,estimators_num+1)
ax.plot(list(X),list(clf.staged_score(X_train,y_train)),label="Traing score")
ax.plot(list(X),list(clf.staged_score(X_test,y_test)),label="Testing score")
ax.set_xlabel("estimator num")
ax.set_ylabel("score")
ax.legend(loc="lower right")
ax.set_ylim(0,1)
ax.set_title("AdaBoostClassifier with Gaussian Naive Bayes")
plt.show()
```

调用函数test_AdaBoostClassifier_base_classifier，运行结果如图 10.2 所示。本次测试对比了默认的决策树个体分类器以及高斯分布贝叶斯个体分类器的差别。这里的个体分类器要求满足两个条件：

- ❑ 个体分类器支持带样本训练。
- ❑ 如果algorithm='SAMME.R'，则个体分类器必须支持计算各类别的概率。

如果不支持这两个条件，则程序运行报错。从比较结果来看，由于高斯分布贝叶斯个体分类器本身就是强分类器（即单个分类器的预测准确率已经非常好），所以它没有一个明显的预测准确率提升的过程，整体曲线都比较平缓。

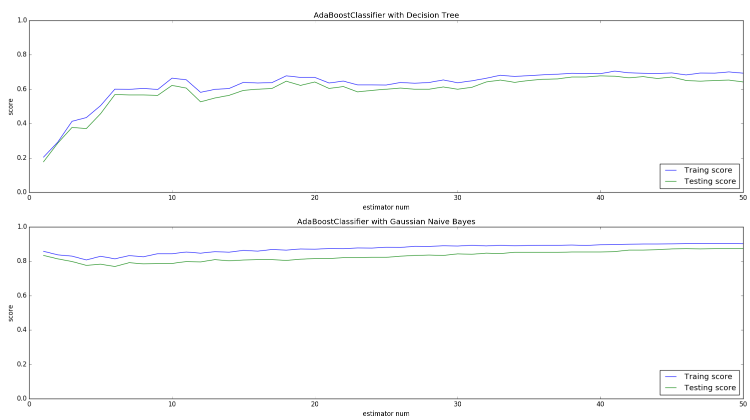


图 10.2 AdaBoostClassifier_base_estimator

下面考察学习率的影响，给出测试函数：

```
def test_AdaBoostClassifier_learning_rate(*data):
    X_train,X_test,y_train,y_test=data
    learning_rates=np.linspace(0.01,1)
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    traing_scores=[]
    testing_scores=[]
    for learning_rate in learning_rates:
        clf=ensemble.AdaBoostClassifier(learning_rate=learning_rate,n_estimators=500)
        clf.fit(X_train,y_train)
        traing_scores.append(clf.score(X_train,y_train))
        testing_scores.append(clf.score(X_test,y_test))
    ax.plot(learning_rates,traing_scores,label="Traing score")
    ax.plot(learning_rates,testing_scores,label="Testing score")
    ax.set_xlabel("learning rate")
    ax.set_ylabel("score")
    ax.legend(loc="best")
    ax.set_title("AdaBoostClassifier")
    plt.show()
```

调用函数test_AdaBoostClassifier_learning_rate，运行结果如图 10.3 所示。当采用默认的SAMME.R算法时，可以看到当学习率较小时，测试准确率和训练准确率随着学习率的增大而缓慢上升。但是当学习率在超过 0.7 之后，随着学习率的上升，迅速下降。根据scikit-learn官方文档的解释如下：

“通常比较小的学习率会要求更多数量的弱学习器从而能保证集成效果。经验表明：比较小的学习率能够带来更小的测试误差。因此推荐学习率小于等于 0.1，同时选择一个很大的n_stimators（因为算法可能在训练误差到达一定程度时，就停止继续训练，这称为‘早停’early stopping，此时真实的个体学习器数量会小于n_stimators）。”

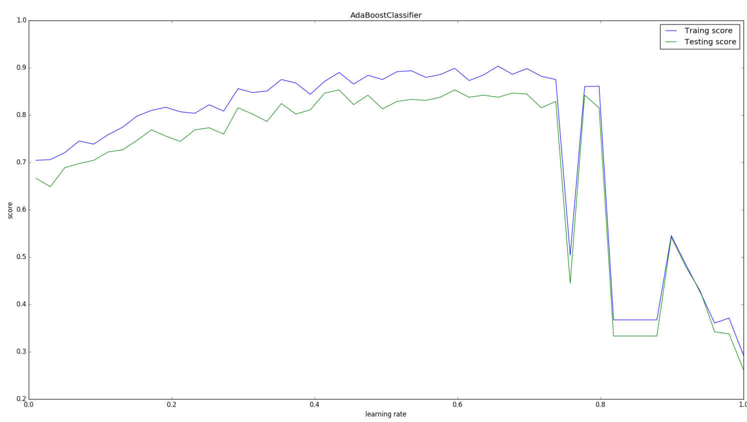


图 10.3 AdaBoostClassifier_learningrate

当将

```
clf=ensemble.AdaBoostClassifier(learning_rate=learning_rate,n_estimators=500)
```

修改为

```
clf=ensemble.AdaBoostClassifier(learning_rate=learning_rate,n_estimators=500,
        algorithm='SAMME')
```

时，即采用SAMME算法时，运行结果如图 10.4 所示。可以看到采用SAMME算法时，测试准确率和训练准确率随着学习率的变化是先上升后稳定的。这是因为当学习率较小时，理论上需要更多数量的弱学习器，但这个例子中固定了弱学习器的数量为 500，相对于低水平的学习率来讲，数量不够。随着学习率的上升，达到良好性能所需要的弱学习器数量的需求也在下降。

采用SAMME算法在学习率上升到 1 附近时并未产生性能下降的问题，这是由于SAMME算法和SAMME.R算法的原理不同所致。

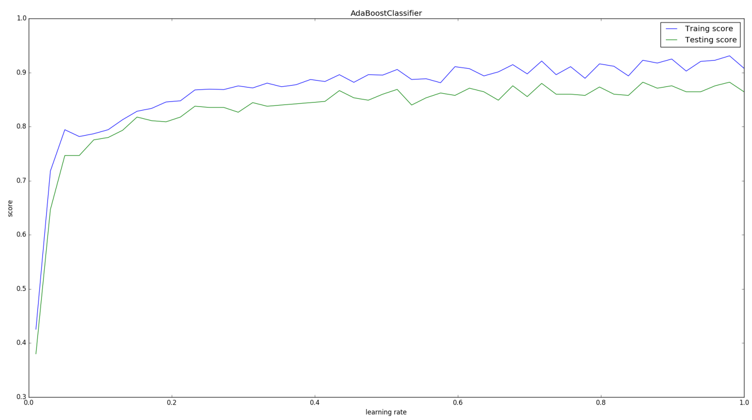


图 10.4　AdaBoostClassifier_learningrate_SAMME

最后考察algorithm的影响，给出测试函数：

```
def test_AdaBoostClassifier_algorithm(*data):
    X_train,X_test,y_train,y_test=data
    algorithms=['SAMME.R','SAMME']
    fig=plt.figure()
    learning_rates=[0.05,0.1,0.5,0.9]
    for i,learning_rate in enumerate(learning_rates):
        ax=fig.add_subplot(2,2,i+1)
        for i ,algorithm in enumerate(algorithms):
            clf=ensemble.AdaBoostClassifier(learning_rate=learning_rate,
                algorithm=algorithm)
            clf.fit(X_train,y_train)
            ## 绘图
            estimators_num=len(clf.estimators_)
```

```
        X=range(1,estimators_num+1)
        ax.plot(list(X),list(clf.staged_score(X_train,y_train)),
            label="%s:Traing score"%algorithms[i])
        ax.plot(list(X),list(clf.staged_score(X_test,y_test)),
            label="%s:Testing score"%algorithms[i])
    ax.set_xlabel("estimator num")
    ax.set_ylabel("score")
    ax.legend(loc="lower right")
    ax.set_title("learing rate:%f"%learning_rate)
fig.suptitle("AdaBoostClassifier")
plt.show()
```

调用函数test_AdaBoostClassifier_algorithm，运行结果如图 10.5 所示。可以看到，当学习率较小时，SAMME.R算法总是预测性能较好。但是当学习率较大时，SAMME.R算法在个体分类器数量较小时预测性能较好，但是个体决策树数量较多时预测性能较差。这是因为SAMME.R算法在个体分类器数量饱和状态下的性能在学习率较大时会迅速下降。

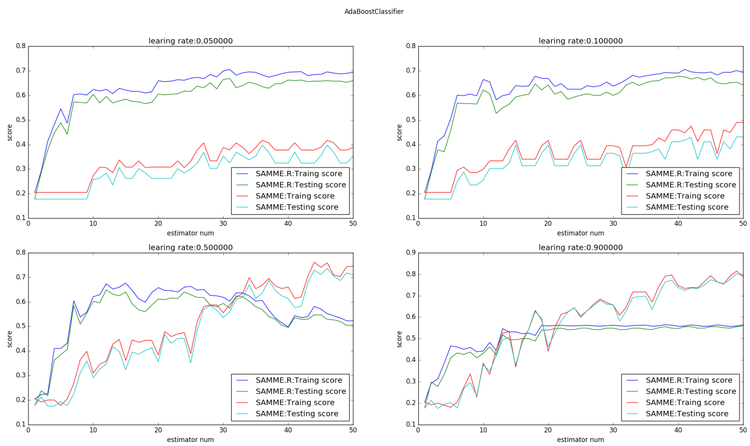


图 10.5 AdaBoostClassifier_algorithm

AdaBoostRegressor 回归器

AdaBoostRegressor是 scikit-learn提供的AdaBoost回归器，其原型为

```
class sklearn.ensemble.AdaBoostRegressor(base_estimator=None, n_estimators=50, \
learning_rate=1.0, loss='linear', random_state=None)
```

参数

- base_estimator：一个基础回归器对象。默认为DecisionTreeRegressor。该基础分类器必须支持带样本权重的学习。
- n_estimators：一个整数，指定基础回归器的数量（默认为 50）。当然如果训练集已经完美地训练好了，可能算法会提前停止。此时基础回归器数量少于该值。

- learning_rate:浮点数,默认为 1。它用于减少每一步的步长,防止步长太大而跨过了极值点。通常learning_rate越小，则需要的基础回归器数量会越多，因此在learning_rate和n_estimators之间会有所折中。learning_rate就是下式中的 ν

$$H_m(\vec{x}) = H_{m-1}(\vec{x}) + \nu\alpha_m h_m(\vec{x})$$

- loss：一个字符串。指定了损失函数。可以为
  - 'linear':线性损失函数（默认）。
  - 'square'：平方损失函数。
  - 'exponential'：指数损失函数。
- random_state：一个整数，或者一个RandomState实例，或者None。
  - 如果为整数，则它指定了随机数生成器的种子。
  - 如果为RandomState实例，则指定了随机数生成器。
  - 如果为None，则使用默认的随机数生成器。

属性

- estimators_：回归器的实例的数组。它存放的是所有训练过的基础回归器。
- estimator_weights_：一个浮点数的数组，给出了每个基础回归器的权重。
- estimator_errors_：一个浮点数的数组，给出了每个基础回归器的分类误差。
- feature_importances_：一个数组，形状为[n_features]。如果base_estimator支持，则它给出了每个特征的重要性。

方法

- fit(X, y[, sample_weight])：训练模型。
- predict(X)：用模型进行预测，返回预测值。
- score(X,y[,sample_weight])：返回预测性能得分。设预测集为 T_{test}，真实值为 y_i，真实值的均值为 $\bar{y}$，预测值为 $\hat{y}_i$，则

$$score = 1 - \frac{\sum_{T_{test}}(y_i - \hat{y}_i)^2}{(y_i - \bar{y})^2}$$

  - score不超过 1，但是可能为负值（预测效果太差）。
  - score越大，预测性能越好。
- staged_predict(X)：返回一个数组，数组元素依次是每一轮迭代结束时尚未完成的集成回归器的预测值。
- staged_score(X, y[, sample_weight])：返回一个数字，数组元素依次是每一轮迭代结束时尚未完成的集成回归器的预测性能得分。

首先给出使用AdaBoostRegressor的函数：

```
def test_AdaBoostRegressor(*data):
    X_train,X_test,y_train,y_test=data
    regr=ensemble.AdaBoostRegressor()
    regr.fit(X_train,y_train)
    ## 绘图
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    estimators_num=len(regr.estimators_)
    X=range(1,estimators_num+1)
    ax.plot(list(X),list(regr.staged_score(X_train,y_train)),label="Traing score")
    ax.plot(list(X),list(regr.staged_score(X_test,y_test)),label="Testing score")
    ax.set_xlabel("estimator num")
    ax.set_ylabel("score")
    ax.legend(loc="best")
    ax.set_title("AdaBoostRegressor")
    plt.show()
```

调用该函数

```
X_train,X_test,y_train,y_test=load_data_regression()
test_AdaBoostRegressor(X_train,X_test,y_train,y_test)
```

运行结果如图 10.6 所示。可以看到随着算法的推进，每一轮迭代都产生一个新的个体分类器被集成。此时集成分类器的训练误差和测试误差都在下降（对应的就是训练准确率和测试准确率上升）。

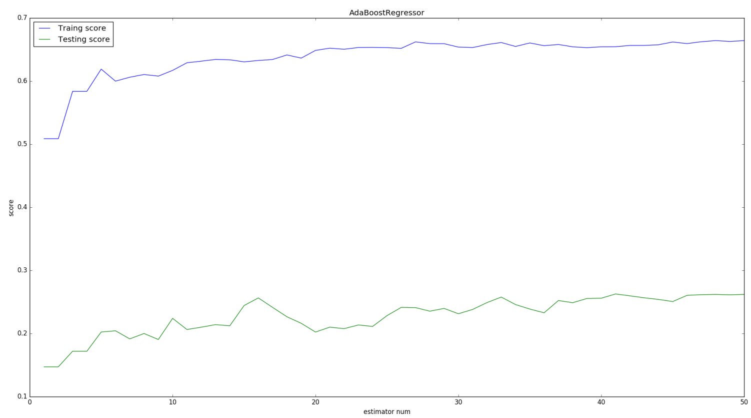


图 10.6 AdaBoostRegressor

下面考察不同类型的个体分类器的影响，给出测试函数：

```
def test_AdaBoostRegressor_base_regr(*data):
    from sklearn.svm import  LinearSVR
    X_train,X_test,y_train,y_test=data
```

```
    fig=plt.figure()
    regrs=[ensemble.AdaBoostRegressor(),
        ensemble.AdaBoostRegressor(base_estimator=LinearSVR(epsilon=0.01,C=100))]
    labels=["Decision Tree Regressor","Linear SVM Regressor"]
    for i ,regr in enumerate(regrs):
        ax=fig.add_subplot(2,1,i+1)
        regr.fit(X_train,y_train)
        ## 绘图
        estimators_num=len(regr.estimators_)
        X=range(1,estimators_num+1)
        ax.plot(list(X),list(regr.staged_score(X_train,y_train)),label="Traing score")
        ax.plot(list(X),list(regr.staged_score(X_test,y_test)),label="Testing score")
        ax.set_xlabel("estimator num")
        ax.set_ylabel("score")
        ax.legend(loc="lower right")
        ax.set_ylim(-1,1)
        ax.set_title("Base_Estimator:%s"%labels[i])
    plt.suptitle("AdaBoostRegressor")
    plt.show()
```

调用函数test_AdaBoostRegressor_base_regr，运行结果如图 10.7 所示。本次测试对比的个体回归器分别为：默认的决策树数目以及线性支持回归器。从比较结果来看，由于线性支持回归器本身就是强回归器（即单个回归器的得分已经很好），所以它没有一个明显的得分提升的过程，整体曲线都比较平缓。可以观察到：使用线性支持回归器时，发生了早停early stopping现象。默认的个体回归器数量为 50，但是这里只有 13 个，因为训练误差满足条件的时候，迭代提前终止。

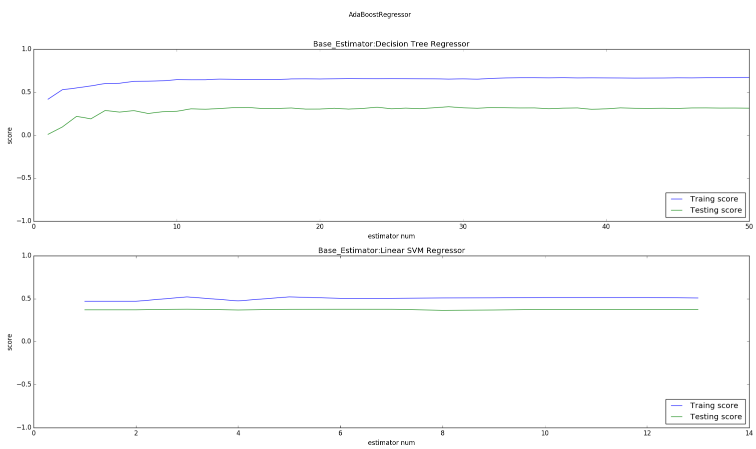


图 10.7 AdaBoostRegressor_base_regr

下面考察学习率的影响。给出测试函数：

```
def test_AdaBoostRegressor_learning_rate(*data):
    X_train,X_test,y_train,y_test=data
```

```
    learning_rates=np.linspace(0.01,1)
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    traing_scores=[]
    testing_scores=[]
    for learning_rate in learning_rates:
        regr=ensemble.AdaBoostRegressor(learning_rate=learning_rate,n_estimators=500)
        regr.fit(X_train,y_train)
        traing_scores.append(regr.score(X_train,y_train))
        testing_scores.append(regr.score(X_test,y_test))
    ax.plot(learning_rates,traing_scores,label="Traing score")
    ax.plot(learning_rates,testing_scores,label="Testing score")
    ax.set_xlabel("learning rate")
    ax.set_ylabel("score")
    ax.legend(loc="best")
    ax.set_title("AdaBoostRegressor")
    plt.show()
```

调用函数test_AdaBoostRegressor_learning_rate，运行结果如图 10.8 所示。可以看到当学习率较大时，预测得分和训练得分都比较稳定。学习率较小时的预测得分较高。

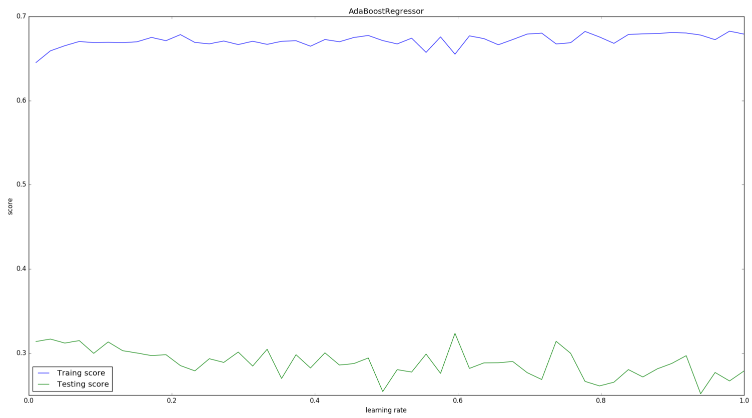


图 10.8 AdaBoostRegressor_learning_rate

最后考察损失函数的影响，给出测试函数：

```
def test_AdaBoostRegressor_loss(*data):
    X_train,X_test,y_train,y_test=data
    losses=['linear','square','exponential']
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    for i ,loss in enumerate(losses):
        regr=ensemble.AdaBoostRegressor(loss=loss,n_estimators=30)
        regr.fit(X_train,y_train)
        ## 绘图
```

```
        estimators_num=len(regr.estimators_)
        X=range(1,estimators_num+1)
        ax.plot(list(X),list(regr.staged_score(X_train,y_train)),
            label="Traing score:loss=%s"%loss)
        ax.plot(list(X),list(regr.staged_score(X_test,y_test)),
            label="Testing score:loss=%s"%loss)
        ax.set_xlabel("estimator num")
        ax.set_ylabel("score")
        ax.legend(loc="lower right")
        ax.set_ylim(-1,1)
    plt.suptitle("AdaBoostRegressor")
    plt.show()
```

调用函数test_AdaBoostRegressor_loss，运行结果如图 10.9 所示。可以看到不同的损失函数对训练得分和测试得分影响不大。

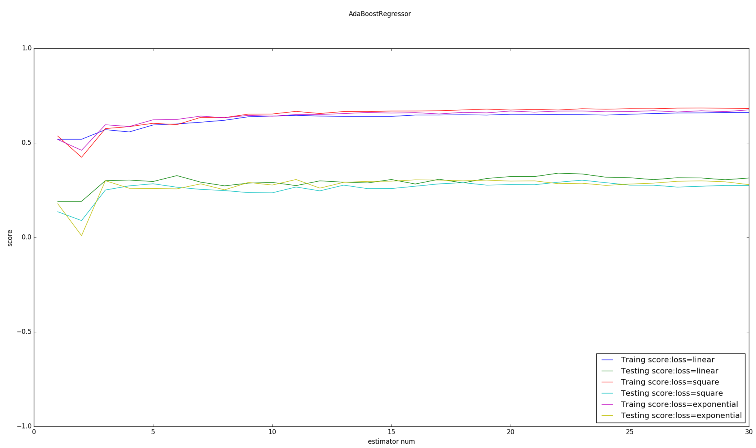


图 10.9 AdaBoostRegressor_loss

10.3.2 Gradient Tree Boosting

scikit-learn基于梯度提升树算法提供了两个模型：

- ❑ GradientBoostingClassifier即GBDT，用于分类问题；
- ❑ GradientBoostingRegressor即GBRT，用于回归问题。

GradientBoostingClassifier 梯度提升决策树

GradientBoostingClassifier是scikit-learn提供的梯度提升决策树，其原型为：

```
class sklearn.ensemble.GradientBoostingClassifier(loss='deviance', learning_rate=0.1,
 n_estimators=100, subsample=1.0, min_samples_split=2, min_samples_leaf=1,
```

```
 min_weight_fraction_leaf=0.0, max_depth=3, init=None, random_state=None,
max_features=None, verbose=0, max_leaf_nodes=None, warm_start=False, presort='auto')
```

参数

- loss：一个字符串，指定损失函数。可以为
  - 'deviance'（默认值）：此时损失函数为对数损失函数：$L(Y, P(Y/X)) = -\log P(Y/X)$。
  - 'exponential'：此时使用指数损失函数。
- n_estimators：一个整数，指定基础决策树的数量（默认为 100）。GBDT对过拟合有很好的鲁棒性，因此该值越大越好。
- learning_rate：浮点数，默认为 0.1。它用于减少每一步的步长，防止步长太大而跨过了极值点。通常learning_rate越小，则需要的基础决策树数量会越多，因此在learning_rate和n_estimators之间会有所折中。learning_rate就是下式中的 ν。

$$H_m(\vec{x}) = H_{m-1}(\vec{x}) + \nu\alpha_m h_m(\vec{x})$$

- max_depth：一个整数或者None，指定了每个基础决策树模型的max_depth参数。调整该参数可以获得最佳性能。如果max_leaf_nodes不是None，则忽略本参数。
- min_samples_split：一个整数，指定了每个基础决策树模型的min_samples_split参数。
- min_samples_leaf：一个整数，指定了每个基础决策树模型的min_samples_leaf参数。
- min_weight_fraction_leaf：一个浮点数，指定了每个基础决策树模型的 min_weight_fraction_leaf 参数。
- subsample：一个浮点数，指定了提取原始训练集中的一个子集用于训练基础决策树。该参数就是子集占原始训练集的大小，大于 0，小于 1.0。
  - 如果 subsample小于 1.0，则梯度提升决策树模型就是随机梯度提升决策树，此时会减少方差但是提高了偏差，它会影响n_estimators参数。
- max_features：一个整数或者浮点数或者字符串或者None，指定了每个基础决策树模型的max_features参数。
  - 如果 max_features< n_features，则会减少方差但是提高了偏差。
- max_leaf_nodes：为整数或者None，指定了每个基础决策树模型的max_leaf_nodes参数。
- init：一个基础分类器对象或者None，该分类器对象用于执行初始的预测。如果为None，则使用loss.init_estimator。
- verbose：一个整数。如果为 0 则不输出日志信息；如果为 1 则每隔一段时间打印一次日志信息；如果大于 1，则打印日志信息更频繁。
- warm_start：布尔值。当为True时，则继续使用上一次训练的结果。否则重新开始训练。
- random_state：一个整数，或者一个RandomState实例，或者None。

- ❍ 如果为整数，则它指定了随机数生成器的种子。
- ❍ 如果为RandomState实例，则指定了随机数生成器。
- ❍ 如果为None，则使用默认的随机数生成器。

- ❑ presort：一个布尔值或者'auto'，指定了每个基础决策树模型的presort参数。

属性

- ❑ feature_importances_：一个数组，给出了每个特征的重要性（值越高，重要性越大）。
- ❑ oob_improvement_：一个数组，给出了每增加一棵基础决策树，在包外估计（即测试集）的损失函数的改善情况（即损失函数的减少值）。
- ❑ train_score_：一个数组，给出了每增加一棵基础决策树，在训练集上的损失函数的值。
- ❑ init：初始预测使用的分类器。
- ❑ estimators_：一个数组，给出了每棵基础决策树。

方法

- ❑ fit(X, y[, sample_weight, monitor])：训练模型。其中monitor是一个可调用对象，它在当前迭代过程结束时调用。如果它返回True，则训练过程提前终止。
- ❑ predict(X)：用模型进行预测，返回预测值。
- ❑ predict_log_proba(X)：返回一个数组，数组的元素依次是X预测为各个类别的概率的对数值。
- ❑ predict_proba(X)：返回一个数组，数组的元素依次是X预测为各个类别的概率值。
- ❑ score(X,y[,sample_weight])：返回在(X,y)上预测的准确率(accuracy)。
- ❑ staged_predict(X)：返回一个数组，数组元素依次是每一轮迭代结束时集成分类器的预测值。
- ❑ staged_predict_proba(X)：返回一个二维数组，数组元素依次是每一轮迭代结束时集成分类器预测X为各个类别的概率值。

首先给出使用GradientBoostingClassifier的函数：

```
def test_GradientBoostingClassifier(*data):
    X_train,X_test,y_train,y_test=data
    clf=ensemble.GradientBoostingClassifier()
    clf.fit(X_train,y_train)
    print("Traing Score:%f"%clf.score(X_train,y_train))
    print("Testing Score:%f"%clf.score(X_test,y_test))
```

调用该函数

```
X_train,X_test,y_train,y_test=load_data_classification()
test_GradientBoostingClassifier(X_train,X_test,y_train,y_test)
```

运行结果如下：

```
Traing Score:1.000000
Testing Score:0.962222
```

可以看到GBDT对训练集完美拟合（100%），对测试集的预测准确率高达 96.2%。

下面考察个体决策树的数量对于GBDT预测性能的影响，给出函数：

```
def test_GradientBoostingClassifier_num(*data):
    X_train,X_test,y_train,y_test=data
    nums=np.arange(1,100,step=2)
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    testing_scores=[]
    training_scores=[]
    for num in nums:
        clf=ensemble.GradientBoostingClassifier(n_estimators=num)
        clf.fit(X_train,y_train)
        training_scores.append(clf.score(X_train,y_train))
        testing_scores.append(clf.score(X_test,y_test))
    ax.plot(nums,training_scores,label="Training Score")
    ax.plot(nums,testing_scores,label="Testing Score")
    ax.set_xlabel("estimator num")
    ax.set_ylabel("score")
    ax.legend(loc="lower right")
    ax.set_ylim(0,1.05)
    plt.suptitle("GradientBoostingClassifier")
    plt.show()
```

调用test_GradientBoostingClassifier_num函数，结果如图 10.10 所示。可以看到，随着个体决策树数量的增长，GBDT的性能很快上升并保持稳定，且对训练集一直能保持完美拟合，对测试集的预测准确率都在 95% 以上。可以看到，梯度提升决策树能够很好地抵抗过拟合。

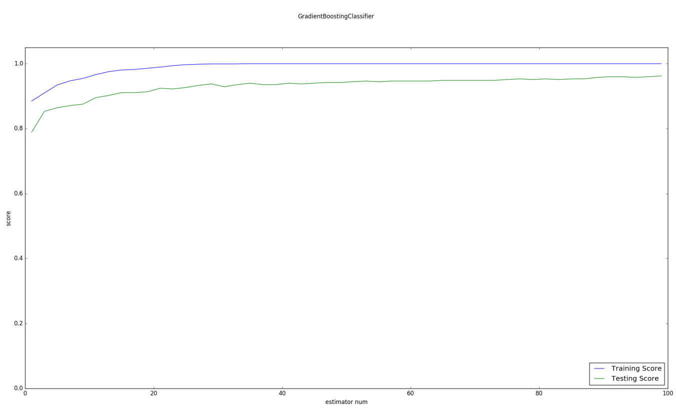


图 10.10　GradientBoostingClassifier_num

下面考察个体决策树的最大树深对于GBDT预测性能的影响，给出函数：

```
def test_GradientBoostingClassifier_maxdepth(*data):
    X_train,X_test,y_train,y_test=data
    maxdepths=np.arange(1,20)
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    testing_scores=[]
    training_scores=[]
    for maxdepth in maxdepths:
        clf=ensemble.GradientBoostingClassifier(max_depth=maxdepth,max_leaf_nodes=None)
        clf.fit(X_train,y_train)
        training_scores.append(clf.score(X_train,y_train))
        testing_scores.append(clf.score(X_test,y_test))
    ax.plot(maxdepths,training_scores,label="Training Score")
    ax.plot(maxdepths,testing_scores,label="Testing Score")
    ax.set_xlabel("max_depth")
    ax.set_ylabel("score")
    ax.legend(loc="lower right")
    ax.set_ylim(0,1.05)
    plt.suptitle("GradientBoostingClassifier")
    plt.show()
```

调用test_GradientBoostingClassifier_maxdepth函数，结果如图 10.11 所示。由于这里分类样本为 1797 个，训练集占 $\frac{3}{4}$ (约 1350 个)。因此这里将子树的最大深度设为 20。可以看到随着个体决策树的最大深度的增大，GBDT对训练集的拟合一直都较好；但是GBDT对于预测集的拟合有所波动。

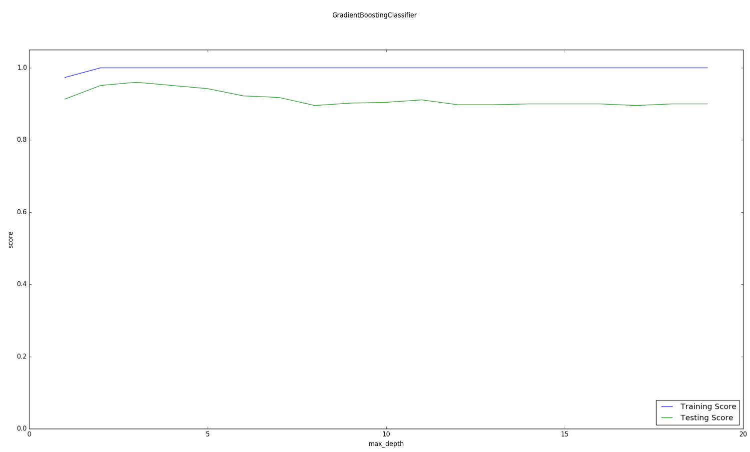


图 10.11 GradientBoostingClassifier_maxdepth

接着考察学习率对于GBDT的预测性能的影响，给出函数：

```
def test_GradientBoostingClassifier_learning(*data):
    X_train,X_test,y_train,y_test=data
```

```
    learnings=np.linspace(0.01,1.0)
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    testing_scores=[]
    training_scores=[]
    for learning in learnings:
        clf=ensemble.GradientBoostingClassifier(learning_rate=learning)
        clf.fit(X_train,y_train)
        training_scores.append(clf.score(X_train,y_train))
        testing_scores.append(clf.score(X_test,y_test))
    ax.plot(learnings,training_scores,label="Training Score")
    ax.plot(learnings,testing_scores,label="Testing Score")
    ax.set_xlabel("learning_rate")
    ax.set_ylabel("score")
    ax.legend(loc="lower right")
    ax.set_ylim(0,1.05)
    plt.suptitle("GradientBoostingClassifier")
    plt.show()
```

调用test_GradientBoostingClassifier_learning函数，结果如图 10.12 所示。可以看到GBDT的预测准确率对于学习率总体上比较平稳。由于学习率必须大于 0，所以在学习率比较小时，预测准确率有一个上升阶段。

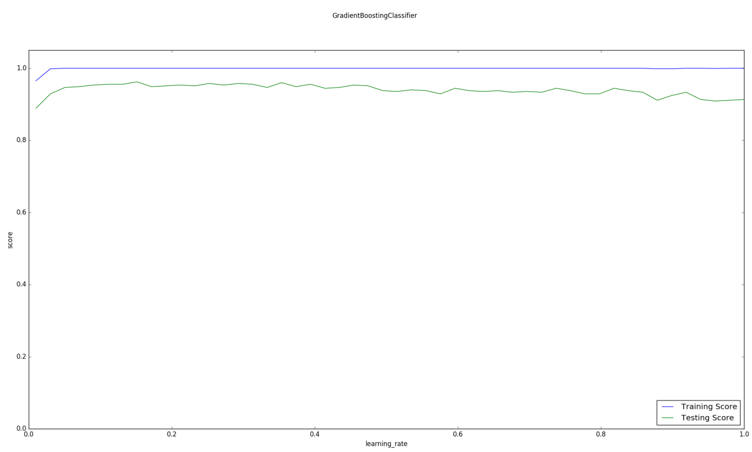


图 10.12 GradientBoostingClassifier_learning

然后考察subsample的影响。当subsample!=1时，就是随机梯度提升树。给出测试函数：

```
def test_GradientBoostingClassifier_subsample(*data):
    X_train,X_test,y_train,y_test=data
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    subsamples=np.linspace(0.01,1.0)
    testing_scores=[]
```

```
    training_scores=[]
    for subsample in subsamples:
        clf=ensemble.GradientBoostingClassifier(subsample=subsample)
        clf.fit(X_train,y_train)
        training_scores.append(clf.score(X_train,y_train))
        testing_scores.append(clf.score(X_test,y_test))
    ax.plot(subsamples,training_scores,label="Training Score")
    ax.plot(subsamples,testing_scores,label="Training Score")
    ax.set_xlabel("subsample")
    ax.set_ylabel("score")
    ax.legend(loc="lower right")
    ax.set_ylim(0,1.05)
    plt.suptitle("GradientBoostingClassifier")
    plt.show()
```

调用函数test_GradientBoostingClassifier_subsample，运行结果如图 10.13 所示。从图中可以看到，当subsample较小时GBDT预测性能较差，原因是每次随机挑选的训练样本太少，抛弃了大量的样本信息。

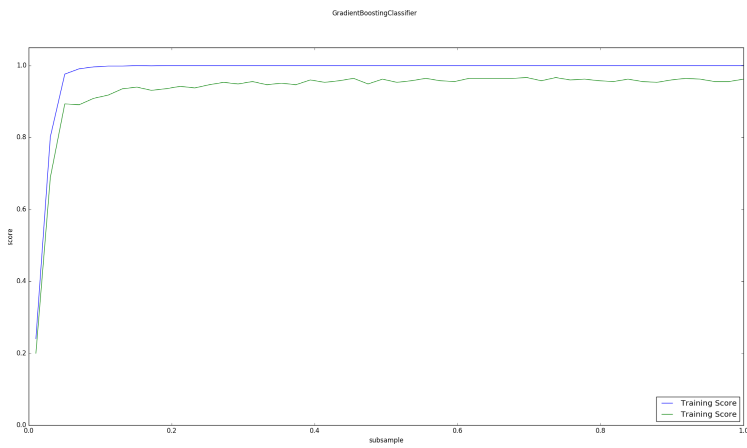


图 10.13　GradientBoostingClassifier_subsample

最后考察 max_features参数的影响。当max_features取浮点数时，它的值在(0,1]之间。如果max_features!=1.0，则每次决策树的特征选取的集合是原来特征集合的一个子集。这里给出测试函数：

```
def test_GradientBoostingClassifier_max_features(*data):
    X_train,X_test,y_train,y_test=data
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    max_features=np.linspace(0.01,1.0)
    testing_scores=[]
    training_scores=[]
    for features in max_features:
```

```
            clf=ensemble.GradientBoostingClassifier(max_features=features)
            clf.fit(X_train,y_train)
            training_scores.append(clf.score(X_train,y_train))
            testing_scores.append(clf.score(X_test,y_test))
    ax.plot(max_features,training_scores,label="Training Score")
    ax.plot(max_features,testing_scores,label="Training Score")
    ax.set_xlabel("max_features")
    ax.set_ylabel("score")
    ax.legend(loc="lower right")
    ax.set_ylim(0,1.05)
    plt.suptitle("GradientBoostingClassifier")
    plt.show()
```

调用test_GradientBoostingClassifier_max_features函数，结果如图 10.14 所示。可以看到GBDT对于特征集合的选取不是很敏感。

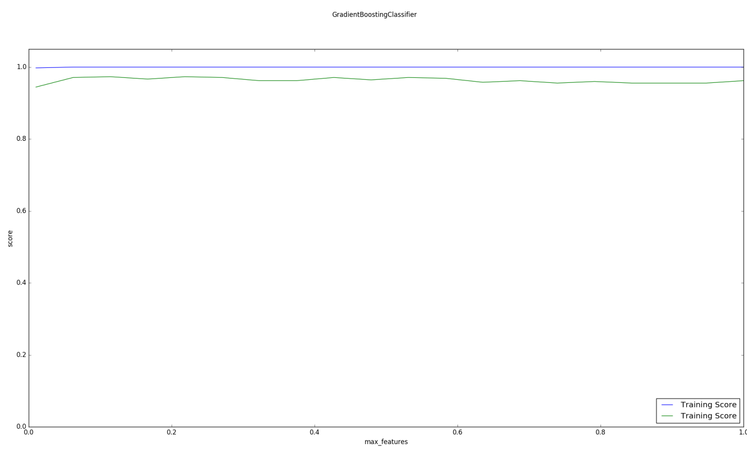


图 10.14　GradientBoostingClassifier_max_features

GradientBoostingRegressor

GradientBoostingRegressor是scikit-learn提供的GBRT模型，其原型为：

```
class sklearn.ensemble.GradientBoostingRegressor(loss='ls', learning_rate=0.1,
 n_estimators=100, subsample=1.0, min_samples_split=2, min_samples_leaf=1,
 min_weight_fraction_leaf=0.0, max_depth=3, init=None, random_state=None,
 max_features=None, alpha=0.9, verbose=0, max_leaf_nodes=None, warm_start=False,
presort='auto')
```

参数

- loss：一个字符串，指定损失函数，可以如下。
    - 'ls'：损失函数为平方损失函数。

- 'lad'：损失函数为绝对值损失函数。
- 'huber'：损失函数为上述两者的结合，通过alpha参数指定比例，该损失函数的定义为

$$L_{Huber} = \begin{cases} \frac{1}{2}(y - f(x))^2, & \text{if} \quad |y - f(x)| \le \alpha \\ \alpha|y - f(x)| - \frac{1}{2}\alpha^2, & \text{else} \end{cases}$$

即误差较小时，采用平方损失；在误差较大时，采用绝对值损失。
- 'quantile'：分位数回归（分位数指的是百分之几），通过alpha参数指定分位数。

- alpha：一个浮点数，只有当loss='huber'或者loss='quantile'时才有效，指定了alpha-quantile。
- n_estimators：一个整数，指定基础回归树的数量（默认为 100 棵）。GBRT对过拟合有很好的鲁棒性，因此该值越大越好。
- learning_rate：浮点数。默认为 0.1。它用于减少每一步的步长，防止步长太大而跨过了极值点。通常learning_rate越小，则需要的基础回归树数量会越多，因此在learning_rate和n_estimators之间会有所折中。learning_rate就是下式中的 ν。

$$H_m(\vec{\boldsymbol{x}}) = H_{m-1}(\vec{\boldsymbol{x}}) + \nu\alpha_m h_m(\vec{\boldsymbol{x}})$$

- max_depth：一个整数或者None，指定了每个基础回归树模型的max_depth参数。调整该参数可以获得最佳性能。如果max_leaf_nodes不是None，则忽略本参数。
- min_samples_split：一个整数，指定了每个基础回归树模型的min_samples_split参数。
- min_samples_leaf：一个整数，指定了每个基础回归树模型的min_samples_leaf参数。
- min_weight_fraction_leaf：一个浮点数，指定了每个基础回归树模型的 min_weight_fraction_leaf 参数。
- subsample：一个浮点数，指定了提取原始训练集中的一个子集用于训练基础回归树。该参数就是子集占原始训练集的大小：大于 0，小于 1.0。
  - 如果 subsample小于 1.0，则梯度提升回归树模型就是随机梯度提升回归树，此时会减少方差但是提高了偏差。它会影响n_estimators参数。
- max_features：一个整数或者浮点数或者字符串或者None，指定了每个基础回归树模型的max_features参数。
  - 如果 max_features< n_features，则会减少方差但是提高了偏差。
- max_leaf_nodes：为整数或者None，指定了每个基础回归树模型的max_leaf_nodes参数。
- init：一个基础回归器对象或者None，该回归器对象用于执行初始的预测。如果为None，则使用loss.init_estimator。
- verbose：一个整数。如果为 0 则不输出日志信息；如果为 1 则每隔一段时间打印一次日志信息；如果大于 1，则打印日志信息更频繁。
- warm_start：布尔值。当为True时，则继续使用上一次训练的结果。否则重新开始训练。
- random_state：一个整数，或者一个RandomState实例，或者None。

  - ❍ 如果为整数，则它指定了随机数生成器的种子。
  - ❍ 如果为RandomState实例，则指定了随机数生成器。
  - ❍ 如果为None，则使用默认的随机数生成器。
- ❑ presort：一个布尔值或者'auto'，指定了每个基础回归树模型的presort参数。

属性

- ❑ feature_importances_：一个数组，给出了每个特征的重要性（值越高，重要性越大）。
- ❑ oob_improvement_：一个数组，给出每增加一棵基础回归树，在包外估计（即测试集）的损失函数的改善情况（即损失函数的减少值）。
- ❑ train_score_：一个数组，给出了每增加一棵基础回归树，在训练集上的损失函数的值。
- ❑ init：初始预测使用的回归器。
- ❑ estimators_：一个数组，给出了每棵基础回归树。

方法

- ❑ fit(X, y[, sample_weight, monitor])：训练模型。其中monitor是一个可调用对象，它在当前迭代过程结束时调用。如果它返回True，则训练过程提前终止。
- ❑ predict(X)：用模型进行预测，返回预测值。
- ❑ predict_log_proba(X)：返回一个数组，数组的元素依次是X预测为各个类别的概率的对数值。
- ❑ predict_proba(X)：返回一个数组，数组的元素依次是X预测为各个类别的概率值。
- ❑ score(X,y[,sample_weight])：返回预测性能得分。设预测集为 T_{test}，真实值为 y_i，真实值的均值为 $\bar{y}$，预测值为 $\hat{y}_i$ 时的得分如下。

$$score = 1 - \frac{\sum_{T_{test}}(y_i - \hat{y}_i)^2}{(y_i - \bar{y})^2}$$

  - ❍ score不超过 1，但是可能为负值（预测效果太差）。
  - ❍ score越大，预测性能越好。
- ❑ staged_predict(X)：返回一个数组，数组元素依次是每一轮迭代结束时集成回归器的预测值。

首先给出使用GradientBoostingRegressor的函数：

```
def test_GradientBoostingRegressor(*data):
    X_train,X_test,y_train,y_test=data
    regr=ensemble.GradientBoostingRegressor()
    regr.fit(X_train,y_train)
    print("Training score:%f"%regr.score(X_train,y_train))
    print("Testing score:%f"%regr.score(X_test,y_test))
```

调用该函数

```
X_train,X_test,y_train,y_test=load_data_regression()
test_GradientBoostingRegressor(X_train,X_test,y_train,y_test)
```

运行结果如下。

```
Training score:0.878471
Testing score:0.223260
```

下面考察个体回归树的数量对于GBRT预测性能的影响，给出函数：

```
def test_GradientBoostingRegressor_num(*data):
    X_train,X_test,y_train,y_test=data
    nums=np.arange(1,200,step=2)
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    testing_scores=[]
    training_scores=[]
    for num in nums:
        regr=ensemble.GradientBoostingRegressor(n_estimators=num)
        regr.fit(X_train,y_train)
        training_scores.append(regr.score(X_train,y_train))
        testing_scores.append(regr.score(X_test,y_test))
    ax.plot(nums,training_scores,label="Training Score")
    ax.plot(nums,testing_scores,label="Testing Score")
    ax.set_xlabel("estimator num")
    ax.set_ylabel("score")
    ax.legend(loc="lower right")
    ax.set_ylim(0,1.05)
    plt.suptitle("GradientBoostingRegressor")
    plt.show()
```

调用test_GradientBoostingRegressor_num函数，结果如图 10.15 所示。可以看到，随着个体回归树数量的增长，GBRT的性能对于训练集的拟合一直在提高；但是对于测试集的预测得分先快速上升后基本上缓缓下降。

下面考察个体回归树的最大树深对于集成回归器预测性能的影响，给出函数：

```
def test_GradientBoostingRegressor_maxdepth(*data):
    X_train,X_test,y_train,y_test=data
    maxdepths=np.arange(1,20)
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    testing_scores=[]
    training_scores=[]
    for maxdepth in maxdepths:
        regr=ensemble.GradientBoostingRegressor(max_depth=maxdepth,max_leaf_nodes=None)
```

```
        regr.fit(X_train,y_train)
        training_scores.append(regr.score(X_train,y_train))
        testing_scores.append(regr.score(X_test,y_test))
    ax.plot(maxdepths,training_scores,label="Training Score")
    ax.plot(maxdepths,testing_scores,label="Testing Score")
    ax.set_xlabel("max_depth")
    ax.set_ylabel("score")
    ax.legend(loc="lower right")
    ax.set_ylim(-1,1.05)
    plt.suptitle("GradientBoostingRegressor")
    plt.show()
```

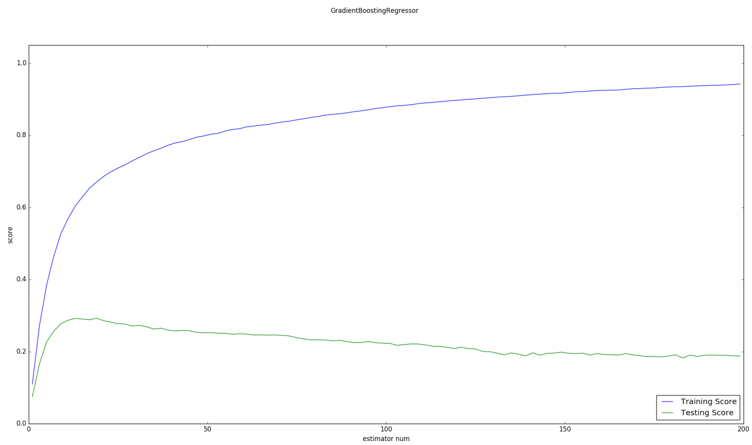


图 10.15　GradientBoostingRegressor_num

调用test_GradientBoostingRegressor_maxdepth函数，结果如图 10.16 所示。可以看出GBRT对测试集的预测得分随着 max_depth缓慢下降后保持平稳。

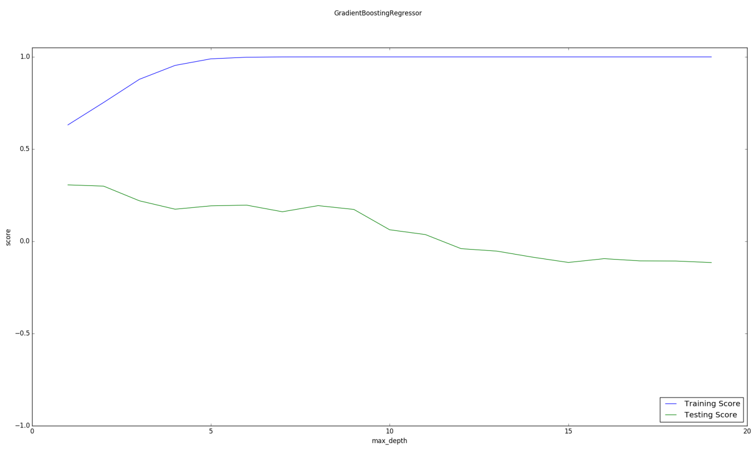


图 10.16　GradientBoostingRegressor_maxdepth

接着考察学习率对于GBRT的预测性能的影响，给出函数：

```
def test_GradientBoostingRegressor_learning(*data):
    X_train,X_test,y_train,y_test=data
    learnings=np.linspace(0.01,1.0)
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    testing_scores=[]
    training_scores=[]
    for learning in learnings:
        regr=ensemble.GradientBoostingRegressor(learning_rate=learning)
        regr.fit(X_train,y_train)
        training_scores.append(regr.score(X_train,y_train))
        testing_scores.append(regr.score(X_test,y_test))
    ax.plot(learnings,training_scores,label="Training Score")
    ax.plot(learnings,testing_scores,label="Testing Score")
    ax.set_xlabel("learning_rate")
    ax.set_ylabel("score")
    ax.legend(loc="lower right")
    ax.set_ylim(-1,1.05)
    plt.suptitle("GradientBoostingRegressor")
    plt.show()
```

调用test_GradientBoostingRegressor_learning函数，结果如图 10.17 所示。可以看出GBRT对测试集的预测得分随着学习率的增长呈现缓慢震荡下降。

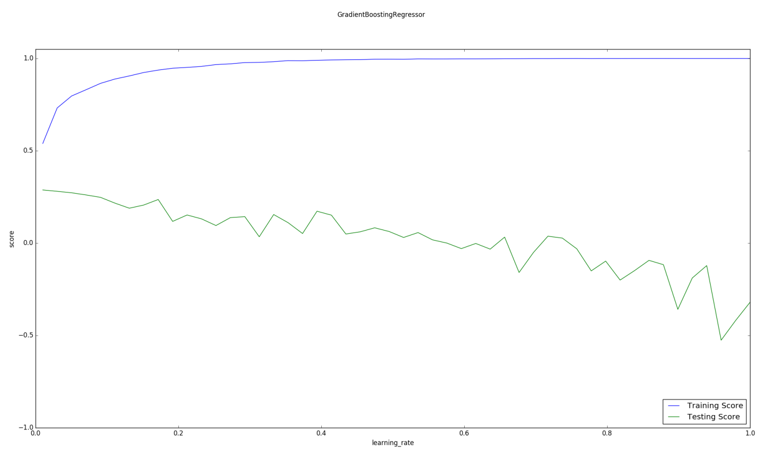


图 10.17　GradientBoostingRegressor_learning

接着考察subsample的影响。当subsample!=1时，就是随机梯度回归树，给出测试函数：

```
def test_GradientBoostingRegressor_subsample(*data):
    X_train,X_test,y_train,y_test=data
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
```

```
    subsamples=np.linspace(0.01,1.0,num=20)
    testing_scores=[]
    training_scores=[]
    for subsample in subsamples:
            regr=ensemble.GradientBoostingRegressor(subsample=subsample)
            regr.fit(X_train,y_train)
            training_scores.append(regr.score(X_train,y_train))
            testing_scores.append(regr.score(X_test,y_test))
    ax.plot(subsamples,training_scores,label="Training Score")
    ax.plot(subsamples,testing_scores,label="Training Score")
    ax.set_xlabel("subsample")
    ax.set_ylabel("score")
    ax.legend(loc="lower right")
    ax.set_ylim(-1,1.05)
    plt.suptitle("GradientBoostingRegressor")
    plt.show()
```

调用函数test_GradientBoostingRegressor_subsample，运行结果如图 10.18 所示。从图中可以看到，在本问题中，subsample对GBRT预测影响不大，它主要对GBRT的训练集拟合能力起作用（即主要影响了训练数据集的误差）。

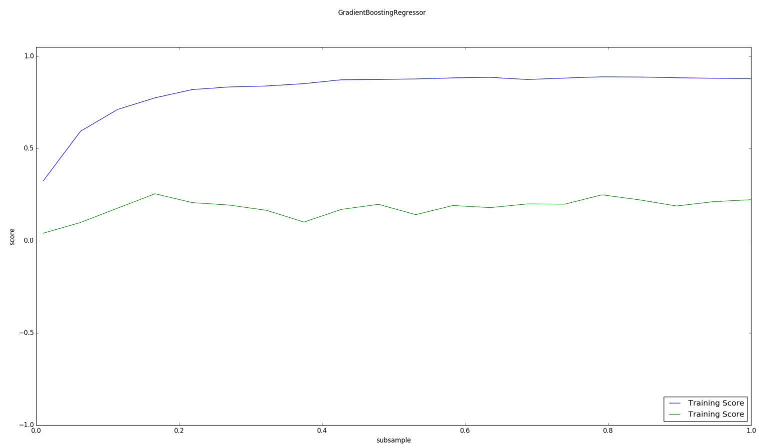


图 10.18 GradientBoostingRegressor_subsample

然后考察损失函数的影响，给出测试函数：

```
def test_GradientBoostingRegressor_loss(*data):
    X_train,X_test,y_train,y_test=data
    fig=plt.figure()
    nums=np.arange(1,200,step=2)
    losses=["ls","lad","huber"]
    ########## 绘制 huber ######
    ax=fig.add_subplot(2,1,1)
    alphas=np.linspace(0.01,1.0,endpoint=False,num=5)
```

```
for alpha in alphas:
        testing_scores=[]
        training_scores=[]
        for num in nums:
                regr=ensemble.GradientBoostingRegressor(n_estimators=num,
                loss='huber',alpha=alpha)
                regr.fit(X_train,y_train)
                training_scores.append(regr.score(X_train,y_train))
                testing_scores.append(regr.score(X_test,y_test))
        ax.plot(nums,training_scores,label="Training Score:alpha=%f"%alpha)
        ax.plot(nums,testing_scores,label="Testing Score:alpha=%f"%alpha)
ax.set_xlabel("estimator num")
ax.set_ylabel("score")
ax.legend(loc="lower right",framealpha=0.4)
ax.set_ylim(0,1.05)
ax.set_title("loss=%huber")
plt.suptitle("GradientBoostingRegressor")
#### 绘制 ls  和 lad
ax=fig.add_subplot(2,1,2)
for loss in ['ls','lad']:
    testing_scores=[]
    training_scores=[]
    for num in nums:
            regr=ensemble.GradientBoostingRegressor(n_estimators=num,loss=loss)
            regr.fit(X_train,y_train)
            training_scores.append(regr.score(X_train,y_train))
            testing_scores.append(regr.score(X_test,y_test))
    ax.plot(nums,training_scores,label="Training Score:loss=%s"%loss)
    ax.plot(nums,testing_scores,label="Testing Score:loss=%s"%loss)
ax.set_xlabel("estimator num")
ax.set_ylabel("score")
ax.legend(loc="lower right",framealpha=0.4)
ax.set_ylim(0,1.05)
ax.set_title("loss=ls,lad")
plt.suptitle("GradientBoostingRegressor")
plt.show()
```

调用函数test_GradientBoostingRegressor_loss，运行结果如图 10.19 所示。这里并没有给出quantile，因为在本问题中，quantile损失函数会出现剧烈震荡。从图中可以发现平方损失函数 ls较好。而huber结合了平方损失函数和绝对值损失函数，性能在两者之间。

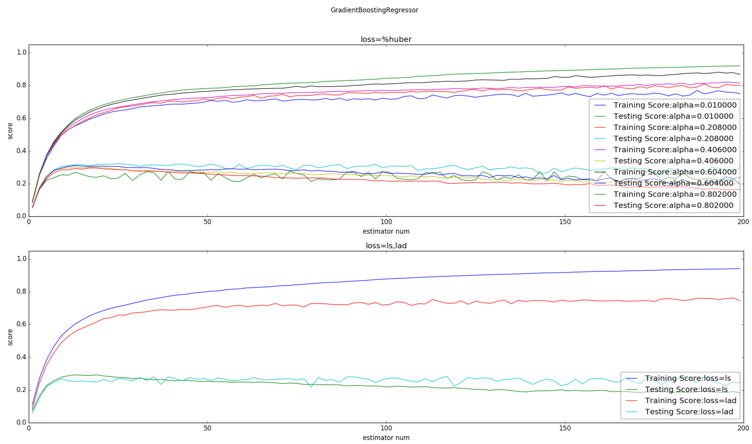


图 10.19 GradientBoostingRegressor_loss

最后考察 max_features参数的影响。当max_features取浮点数时，它的值在(0,1]之间。如果max_features!=1.0，则每次回归树的特征选取的集合是原来特征集合的一个子集。这里给出测试函数：

```
def test_GradientBoostingRegressor_max_features(*data):
    X_train,X_test,y_train,y_test=data
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    max_features=np.linspace(0.01,1.0)
    testing_scores=[]
    training_scores=[]
    for features in max_features:
            regr=ensemble.GradientBoostingRegressor(max_features=features)
            regr.fit(X_train,y_train)
            training_scores.append(regr.score(X_train,y_train))
            testing_scores.append(regr.score(X_test,y_test))
    ax.plot(max_features,training_scores,label="Training Score")
    ax.plot(max_features,testing_scores,label="Training Score")
    ax.set_xlabel("max_features")
    ax.set_ylabel("score")
    ax.legend(loc="lower right")
    ax.set_ylim(0,1.05)
    plt.suptitle("GradientBoostingRegressor")
    plt.show()
```

调用test_GradientBoostingClassifier_max_features函数，结果如图 10.20 所示。可以看到GBRT对于特征集合的选取不是很敏感。

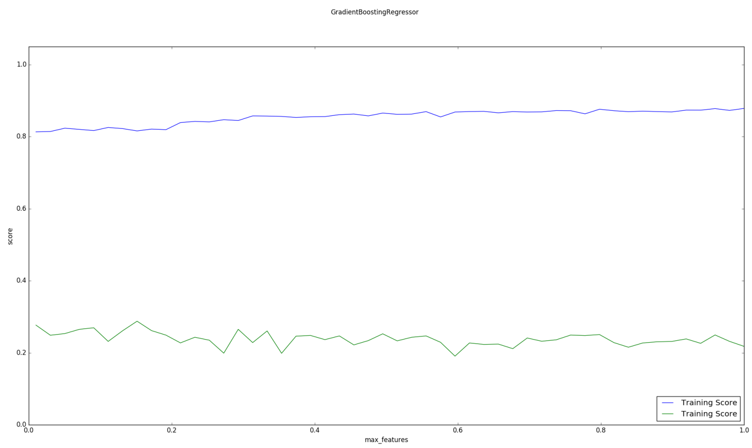


图 10.20 GradientBoostingRegressor_max_features

10.3.3 Random Forest

scikit-learn基于随机森林（Random Forest）算法提供了两个模型：

- ❑ RandomForestClassifier用于分类问题；
- ❑ RandomForestRegressor用于回归问题。

RandomForestClassifier 随机森林分类器

GradientBoostingClassifier是scikit-learn提供的随机森林分类模型，其原型为：

```
class sklearn.ensemble.RandomForestClassifier(n_estimators=10, criterion='gini',
max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0,
 max_features='auto', max_leaf_nodes=None, bootstrap=True, oob_score=False, n_jobs=1,
 random_state=None, verbose=0, warm_start=False, class_weight=None)
```

参数

- ❑ n_estimators：一个整数，指定了随机森林中决策树的数量。
- ❑ criterion：一个字符串，指定了每棵决策树的criterion参数。
- ❑ max_features：一个整数或者浮点数或者字符串或者None，指定了每棵决策树的 max_features 参数。
- ❑ max_depth：一个整数或者None，指定了每棵决策树的max_depth参数。如果max_leaf_nodes不是None，则忽略本参数。
- ❑ min_samples_split：一个整数，指定了每棵决策树的min_samples_split参数。
- ❑ min_samples_leaf：一个整数，指定了每棵决策树的min_samples_leaf参数。
- ❑ min_weight_fraction_leaf：一个浮点数，指定了每棵决策树的min_weight_fraction_leaf参数。

- max_leaf_nodes：为整数或者None，指定了每个基础决策树模型的max_leaf_nodes参数。
- boostrap：为布尔值。如果为True，则使用采样法bootstrap sampling来产生决策树的训练数据集。
- oob_score：为布尔值。如果为True，则使用包外样本来计算泛化误差。
- n_jobs：一个整数，指定并行性。如果为 −1，则表示将训练和预测任务派发到所有CPU上。
- random_state：一个整数，或者一个RandomState实例，或者None。
  - 如果为整数，则它指定了随机数生成器的种子。
  - 如果为RandomState实例，则指定了随机数生成器。
  - 如果为None，则使用默认的随机数生成器。
- verbose：一个整数。如果为 0 则不输出日志信息；如果为 1 则每隔一段时间打印一次日志信息；如果大于 1，则打印日志信息更频繁。
- warm_start: 布尔值。当为True时，则继续使用上一次训练的结果；否则重新开始训练。
- class_weight:一个字典,或者字典的列表,或者字符串'balanced',或者字符串'balanced_subsample'，或者None。
  - 如果为字典，则字典给出了每个分类的权重，如{class_label: weight}。
  - 如果为字符串'balanced'，则每个分类的权重与该分类在样本集中出现的频率成反比。
  - 如果为字符串 'balanced_subsample'，则样本集为采样法bootstrap sampling产生的决策树的训练数据集，每个分类的权重与该分类在采用生成的样本集中出现的频率成反比。
  - 如果为None：则每个分类的权重都为 1。

属性

- estimators_：决策树的实例的数组，它存放的是所有训练过的决策树。
- classes_：一个数组，形状为[n_classes]，为类别标签。
- n_classes_：一个整数，为类别数量。
- n_features_：一个整数，在训练时使用的特征数量。
- n_outputs_：一个整数，在训练时输出的数量。
- feature_importances_：一个数组，形状为[n_features]。如果base_estimator支持，则它给出了每个特征的重要性。
- oob_score_：一个浮点数，训练数据使用包外估计时的得分。

方法

- fit(X, y[, sample_weight])：训练模型。
- predict(X)：用模型进行预测，返回预测值。
- predict_log_proba(X)：返回一个数组，数组的元素依次是X预测为各个类别的概率的对数值。

- predict_proba(X)：返回一个数组，数组的元素依次是X预测为各个类别的概率值。
- score(X,y[,sample_weight])：返回在 (X,y)上预测的准确率 (accuracy)。

首先给出使用RandomForestClassifier的函数：

```
def test_RandomForestClassifier(*data):
    X_train,X_test,y_train,y_test=data
    clf=ensemble.RandomForestClassifier()
    clf.fit(X_train,y_train)
    print("Traing Score:%f"%clf.score(X_train,y_train))
    print("Testing Score:%f"%clf.score(X_test,y_test))
```

调用该函数

```
X_train,X_test,y_train,y_test=load_data_classification()
test_RandomForestClassifier(X_train,X_test,y_train,y_test)
```

运行结果如下：

```
Traing Score:0.999258
Testing Score:0.940000
```

可以看到集成分类器对训练集拟合相当成功（99.9258%），对测试集的预测准确率高达94%。

接下来考察森林中决策树的个数对于总体预测性能的影响。这里给出测试函数：

```
def test_RandomForestClassifier_num(*data):
    X_train,X_test,y_train,y_test=data
    nums=np.arange(1,100,step=2)
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    testing_scores=[]
    training_scores=[]
    for num in nums:
        clf=ensemble.RandomForestClassifier(n_estimators=num)
        clf.fit(X_train,y_train)
        training_scores.append(clf.score(X_train,y_train))
        testing_scores.append(clf.score(X_test,y_test))
    ax.plot(nums,training_scores,label="Training Score")
    ax.plot(nums,testing_scores,label="Testing Score")
    ax.set_xlabel("estimator num")
    ax.set_ylabel("score")
    ax.legend(loc="lower right")
    ax.set_ylim(0,1.05)
    plt.suptitle("RandomForestClassifier")
    plt.show()
```

调用test_RandomForestClassifier_num函数。结果如图 10.21 所示。其分析与 GradientBoostingClassifier 的讨论相同。

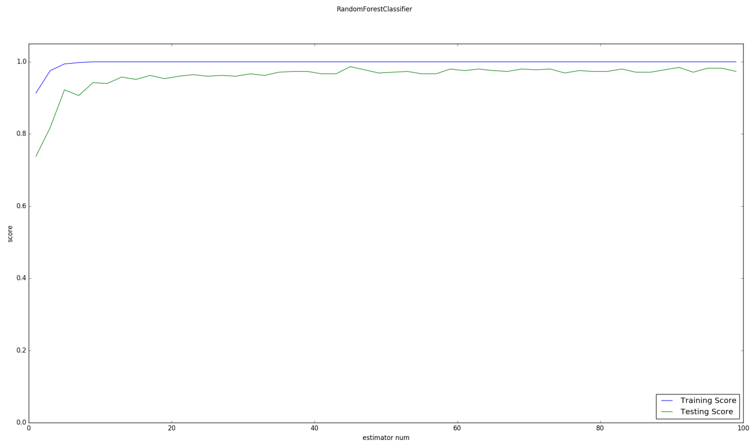


图 10.21 RandomForestClassifier_num

继续考察max_depth参数的影响。这里给出测试函数：

```
def test_RandomForestClassifier_max_depth(*data):
    X_train,X_test,y_train,y_test=data
    maxdepths=range(1,20)
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    testing_scores=[]
    training_scores=[]
    for max_depth in maxdepths:
        clf=ensemble.RandomForestClassifier(max_depth=max_depth)
        clf.fit(X_train,y_train)
        training_scores.append(clf.score(X_train,y_train))
        testing_scores.append(clf.score(X_test,y_test))
    ax.plot(maxdepths,training_scores,label="Training Score")
    ax.plot(maxdepths,testing_scores,label="Testing Score")
    ax.set_xlabel("max_depth")
    ax.set_ylabel("score")
    ax.legend(loc="lower right")
    ax.set_ylim(0,1.05)
    plt.suptitle("RandomForestClassifier")
    plt.show()
```

调用test_RandomForestClassifier_max_depth函数，结果如图 10.22 所示。这里随着树的最大深度的提高，随机森林的预测性能也在提高。这主要有两个原因：

- 决策树的最大深度提高，则每棵树的预测性能也在提高；
- 决策树的最大深度提高，则决策树的多样性也在增大。

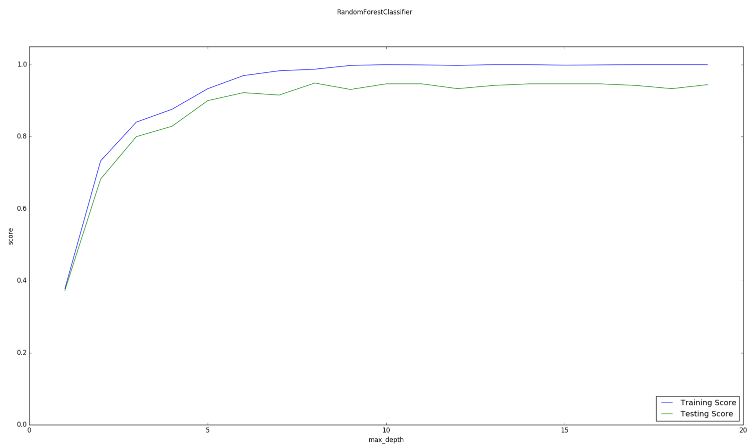


图 10.22 RandomForestClassifier_max_depth

考察 max_features参数的影响。当max_features取浮点数时，它的值在(0,1]之间。如果max_features!=1.0，则每次决策树的特征选取的集合是原来特征集合的一个子集。这里给出测试函数：

```
def test_RandomForestClassifier_max_features(*data):
    X_train,X_test,y_train,y_test=data
    max_features=np.linspace(0.01,1.0)
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    testing_scores=[]
    training_scores=[]
    for max_feature in max_features:
        clf=ensemble.RandomForestClassifier(max_features=max_feature)
        clf.fit(X_train,y_train)
        training_scores.append(clf.score(X_train,y_train))
        testing_scores.append(clf.score(X_test,y_test))
    ax.plot(max_features,training_scores,label="Training Score")
    ax.plot(max_features,testing_scores,label="Testing Score")
    ax.set_xlabel("max_feature")
    ax.set_ylabel("score")
    ax.legend(loc="lower right")
    ax.set_ylim(0,1.05)
    plt.suptitle("RandomForestClassifier")
    plt.show()
```

调用test_RandomForestClassifier_max_features函数，结果如图 10.23 所示。可以看到随机森林对于特征集合的选取不是很敏感。

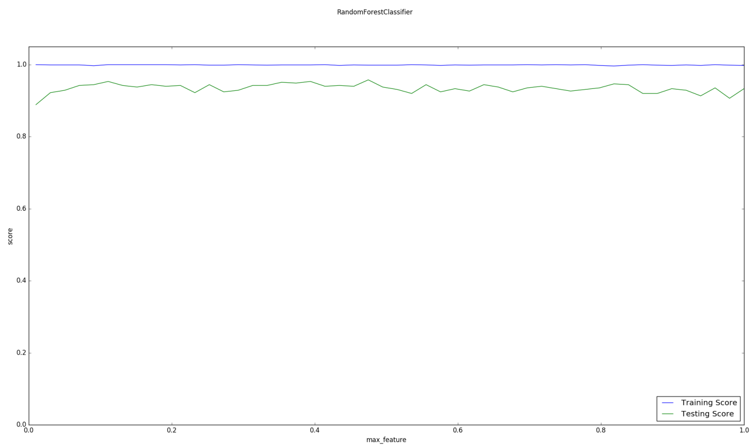


图 10.23 RandomForestClassifier_max_features

RandomForestRegressor 随机森林回归器

RandomForestRegressor是scikit-learn提供的随机森林回归模型，其原型为：

```
class sklearn.ensemble.RandomForestRegressor(n_estimators=10, criterion='mse',
max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0,
 max_features='auto', max_leaf_nodes=None, bootstrap=True, oob_score=False, n_jobs=1,
 random_state=None, verbose=0, warm_start=False
```

参数

- n_estimators：一个整数，指定了随机森林中回归树的数量。
- criterion：一个字符串，指定了每棵回归树的criterion参数。
- max_features：一个整数或者浮点数或者字符串或者None，指定了每棵回归树的 max_features 参数。
- max_depth：一个整数或者None，指定了每棵回归树的max_depth参数。如果max_leaf_nodes不是None，则忽略本参数。
- min_samples_split：一个整数，指定了每棵回归树的min_samples_split参数。
- min_samples_leaf：一个整数，指定了每棵回归树的min_samples_leaf参数。
- min_weight_fraction_leaf：一个浮点数，指定了每棵回归树的min_weight_fraction_leaf参数。
- max_leaf_nodes：为整数或者None，指定了每棵回归树模型的max_leaf_nodes参数。
- boostrap：为布尔值。如果为True，则使用采样法bootstrap sampling来产生回归树的训练数据集。
- oob_score：为布尔值。如果为True，则使用包外样本来计算泛化误差。
- n_jobs：一个整数，指定并行性。如果为 −1，则表示将训练和预测任务派发到所有CPU上。

- random_state：一个整数，或者一个RandomState实例，或者None。
  - 如果为整数，则它指定了随机数生成器的种子。
  - 如果为RandomState实例，则指定了随机数生成器。
  - 如果为None，则使用默认的随机数生成器。
- verbose：一个整数。如果为 0 则不输出日志信息；如果为 1 则每隔一段时间打印一次日志信息；如果大于 1，则打印日志信息更频繁。
- warm_start: 布尔值。当为True时，则继续使用上一次训练的结果。否则重新开始训练。

属性

- estimators_：回归树的实例的数组。它存放的是所有训练过的回归树。
- n_features_：一个整数，在训练时使用的特征数量。
- n_outputs_：一个整数，在训练时，输出的数量。
- feature_importances_：一个数组，形状为[n_features]。如果base_estimator支持，则它给出了每个特征的重要性。
- oob_score_：一个浮点数，训练数据使用包外估计时的得分。
- oob_prediction_：一个数组，训练数据使用包外估计时的预测值。

方法

- fit(X, y[, sample_weight])：训练模型。
- predict(X)：用模型进行预测，返回预测值。
- score(X,y[,sample_weight])：返回预测性能得分。设预测集为 T_{test}，真实值为 y_i，真实值的均值为 $\bar{y}$，预测值为 $\hat{y}_i$ 时的得分为

$$score = 1 - \frac{\sum_{T_{test}}(y_i - \hat{y}_i)^2}{(y_i - \bar{y})^2}$$

  - score不超过 1，但是可能为负值（预测效果太差）。
  - score越大，预测性能越好。

首先给出使用RandomForestRegressor的函数：

```
def test_RandomForestRegressor(*data):
    X_train,X_test,y_train,y_test=data
    regr=ensemble.RandomForestRegressor()
    regr.fit(X_train,y_train)
    print("Traing Score:%f"%regr.score(X_train,y_train))
    print("Testing Score:%f"%regr.score(X_test,y_test))
```

调用该函数

```
X_train,X_test,y_train,y_test=load_data_regression()
test_RandomForestRegressor(X_train,X_test,y_train,y_test)
```

运行结果如下：

```
Traing Score:1.000000
Testing Score:0.009009
```

可以看到集成回归器对训练集拟合相当成功（回归问题中，得分为 1 说明对每个训练样本都准确拟合），对测试集的得分为 0。

接下来考察森林中回归树的棵数对于总体预测性能的影响，这里给出测试函数：

```
def test_RandomForestRegressor_num(*data):
    X_train,X_test,y_train,y_test=data
    nums=np.arange(1,100,step=2)
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    testing_scores=[]
    training_scores=[]
    for num in nums:
        regr=ensemble.RandomForestRegressor(n_estimators=num)
        regr.fit(X_train,y_train)
        training_scores.append(regr.score(X_train,y_train))
        testing_scores.append(regr.score(X_test,y_test))
    ax.plot(nums,training_scores,label="Training Score")
    ax.plot(nums,testing_scores,label="Testing Score")
    ax.set_xlabel("estimator num")
    ax.set_ylabel("score")
    ax.legend(loc="lower right")
    ax.set_ylim(-1,1)
    plt.suptitle("RandomForestRegressor")
    plt.show()
```

调用test_RandomForestRegressor_num函数，结果如图 10.24 所示。可以看到随着回归树的个数的增加，回归森林对于测试集的预测得分先快速上升，然后趋于稳定。

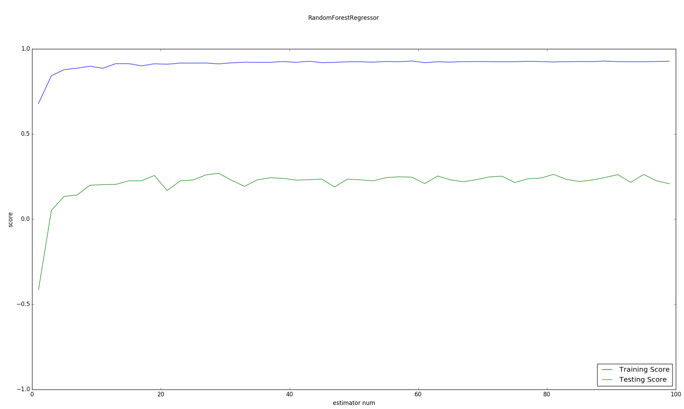


图 10.24　RandomForestRegressor_num

继续考察max_depth参数的影响，这里给出测试函数：

```
def test_RandomForestRegressor_max_depth(*data):
    X_train,X_test,y_train,y_test=data
    maxdepths=range(1,20)
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    testing_scores=[]
    training_scores=[]
    for max_depth in maxdepths:
        regr=ensemble.RandomForestRegressor(max_depth=max_depth)
        regr.fit(X_train,y_train)
        training_scores.append(regr.score(X_train,y_train))
        testing_scores.append(regr.score(X_test,y_test))
    ax.plot(maxdepths,training_scores,label="Training Score")
    ax.plot(maxdepths,testing_scores,label="Testing Score")
    ax.set_xlabel("max_depth")
    ax.set_ylabel("score")
    ax.legend(loc="lower right")
    ax.set_ylim(0,1.05)
    plt.suptitle("RandomForestRegressor")
    plt.show()
```

调用test_RandomForestRegressor_max_depth函数，结果如图 10.25 所示。可以看到回归森林对于测试集的预测得分随着max_depth在震荡，没有显著的趋势。

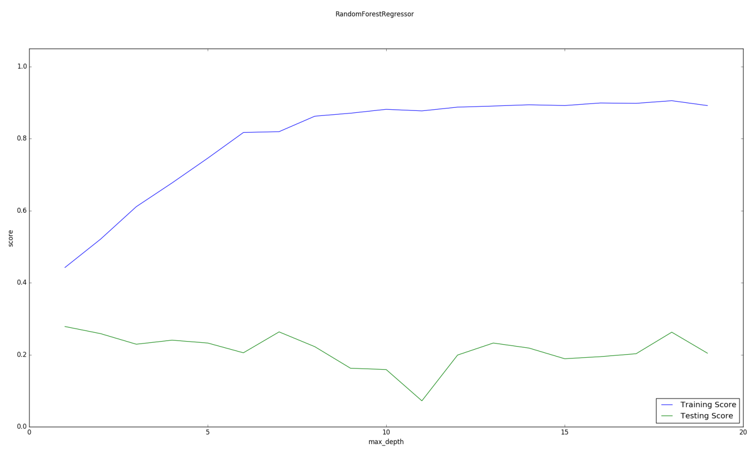


图 10.25 RandomForestRegressor_max_depth

最后考察 max_features参数的影响。当max_features取浮点数时，它的值在(0,1]之间。如果max_features!=1.0，则每次决策树的特征选取的集合是原来特征集合的一个子集。这里给出测试函数：

```
def test_RandomForestRegressor_max_features(*data):
    X_train,X_test,y_train,y_test=data
    max_features=np.linspace(0.01,1.0)
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    testing_scores=[]
    training_scores=[]
    for max_feature in max_features:
        regr=ensemble.RandomForestRegressor(max_features=max_feature)
        regr.fit(X_train,y_train)
        training_scores.append(regr.score(X_train,y_train))
        testing_scores.append(regr.score(X_test,y_test))
    ax.plot(max_features,training_scores,label="Training Score")
    ax.plot(max_features,testing_scores,label="Testing Score")
    ax.set_xlabel("max_feature")
    ax.set_ylabel("score")
    ax.legend(loc="lower right")
    ax.set_ylim(0,1.05)
    plt.suptitle("RandomForestRegressor")
    plt.show()
```

调用test_RandomForestRegressor_max_features函数，结果如图 10.26 所示。可以看到回归森林对于特征集合的选取不是很敏感。

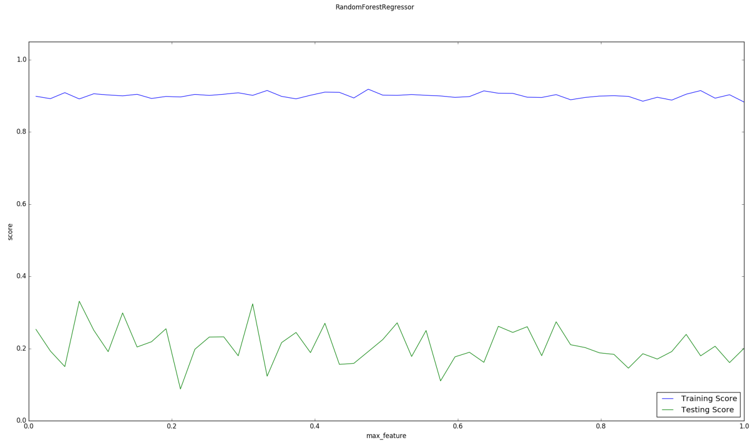


图 10.26 RandomForestRegressor_max_features

10.4 小结

模型	分类模型			回归模型		
	AdaBoost-Classifier	Gradient-Boosting-Classifier	Random-Forest-Classifier	AdaBoost-Regressor	Gradient-Boosting-Regressor	Random-Forest-Regressor
个体学习器数量	越大越好	越大越好	越大越好	越大越好	越大越好	越大越好
个体学习器类型	预测能力越强越好			预测能力越强越好		
学习率	一般较小	一般较小		一般较小	一般较小	
多类分类算法	推荐使用SAMME.R算法					
损失函数				无显著差别	推荐平方误差函数	
max_depth		建议取较小的值	越大越好		建议取较小的值	建议取较小的值
subsample		不宜太小，一般较大			不宜太小，一般较大	
max_features		无显著差别	无显著差别		无显著差别	无显著差别

其中：

- /表示模型没有对应的参数。
- `max_depth`参数对应决策树/回归树的最大深度。
- `subsample`参数对应训练决策树/回归树时，每次随机选取原始训练集的一个子集来训练。
- `max_features`参数对应训练决策树/回归树时，每次随机选取元素特征集的一个子集来训练。

第三篇

机器学习工程篇

第 11 章

数据预处理

11.1 概述

在工程实践中，我们获取的数据因为各种各样的原因，如数据有缺失值、数据有重复值等，需要进行预处理。数据处理没有标准的流程，通常会因为任务的不同、数据集属性的不同而有所不同。这里给出数据预处理的常用流程：

- 去除唯一属性；
- 处理缺失值；
- 属性编码；
- 数据标准化、正则化；
- 特征选择；
- 主成分分析。

其中主成分分析在前述章节详细介绍过，不再复述。这里主要介绍前面的几个常用流程。

注意在这一章中，特征和属性的意义相同，我们不加区分地使用这两个词。

11.2 算法笔记精华

11.2.1 去除唯一属性

在获取的数据集中，经常会遇到唯一属性。这些属性通常是添加的一些id属性，如存放在数据库中自增的主键。这些属性并不能刻画样本自身的分布规律，所以只需要简单地删除这些属性即可。

给定数据集 $D=\{(\vec{\boldsymbol{x}}_1,y_1),(\vec{\boldsymbol{x}}_2,y_2),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\}$ ，其中 $\vec{\boldsymbol{x}}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(d)})^T,i=1,2,\cdots,N$ 。假设 $x^{(t)}$ 为id属性，则去除唯一属性后，有：

$$\hat{\vec{\boldsymbol{x}}}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(t-1)},x_i^{(t+1)},\cdots,x_i^{(d)})^T,i=1,2,\cdots,N$$

11.2.2　处理缺失值的三种方法

数据缺失值产生的原因多种多样，主要分为客观原因和人为原因两种。客观原因是比如数据存储的失败，存储器损坏，机械故障导致某段时间数据未能收集（对于定时数据采集而言）。人为原因是由于人的主观失误、历史局限或有意隐瞒造成的数据缺失，比如，在市场调查中被访人拒绝透露相关问题的答案，或者回答的问题是无效的，数据输入人员失误、漏输了数据。

缺失值的处理有三种方法：

- 直接使用含有缺失值的特征；
- 删除含有缺失值的特征；
- 缺失值补全。

直接使用

对于某些算法可以直接使用含有缺失值的情况。如第 2 章决策树中提到的决策树算法就可以直接使用含有缺失值的数据集。其原理已经在第 2 章决策树 2.2.4 节“连续值和缺失值的处理”讲述，这里不再复述。

删除特征

最简单的办法就是删除含有缺失值的特征。给定数据集 $D=\{(\vec{\boldsymbol{x}}_1,y_1),(\vec{\boldsymbol{x}}_2,y_2),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\}$，其中 $\vec{\boldsymbol{x}}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(d)})^T,i=1,2,\cdots,N$ 。假设 $x^{(t)}$ 属性含有缺失值，则删除该特征，有：

$$\hat{\vec{\boldsymbol{x}}}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(t-1)},x_i^{(t+1)},\cdots,x_i^{(d)})^T,i=1,2,\cdots,N$$

如果 $x^{(t)}$ 含有大量的缺失值，而仅仅包含极少量的有效值，则该方法是最有效的。但是 $x^{(t)}$ 中包含了大量的有效值，则直接删除该特征会丢失大量有效的信息，这是对信息的极大浪费。此时删除该特征不是一个好的办法。

缺失值补全

在缺失值处理的方法中，在实际工程中应用最广泛的是缺失值补全方法，缺失补全的思想是：用最可能的值来插补缺失值，最常见的有以下几种方法：

- ❑ 均值插补；
- ❑ 用同类均值插补；
- ❑ 建模预测；
- ❑ 高维映射；
- ❑ 多重插补；
- ❑ 极大似然估计；
- ❑ 压缩感知及矩阵补全。

11.2.3 常见的缺失值补全方法

均值插补

如果样本属性的距离是可度量的（如身高、体重等），则该属性的缺失值就以该属性有效值的平均值来插补缺失的值。如果样本的属性的距离是不可度量的（如性别、国籍等），则该属性的缺失值就以该属性有效值的众数（出现频率最高的值）来插补缺失的值。

给定数据集 $D=\{(\vec{\boldsymbol{x}}_1,y_1),(\vec{\boldsymbol{x}}_2,y_2),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\}$，其中 $\vec{\boldsymbol{x}}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(d)})^T,i=1,2,\cdots,N$。假设 $x^{(t)}$ 属性含有缺失值，且假设 $(\vec{\boldsymbol{x}}_1,\vec{\boldsymbol{x}}_2,\cdots,\vec{\boldsymbol{x}}_{N1})$ 在 $x^{(t)}$ 属性上含有有效值，$(\vec{\boldsymbol{x}}_{N1+1},\vec{\boldsymbol{x}}_{N1+2},\cdots,\vec{\boldsymbol{x}}_N)$ 在 $x^{(t)}$ 属性为缺失值。提取 $x^{(t)}$ 上的有效值为 $(x_1^{(t)},x_2^{(t)},\cdots,x_{N1}^{(t)})$。

- ❑ 若 $x^{(t)}$ 是可度量的，则：

$$\bar{x}^{(t)}=\frac{1}{N1}\sum_{i=1}^{N1}x_i^{(t)}$$

$$\hat{x}_i^{(t)}=\begin{cases}x_i^{(t)}, & i=1,2,\cdots,N1\\ \bar{x}^{(t)}, & i=N1+1,N1+2,\cdots,N\end{cases}$$

- ❑ 若 $x^{(t)}$ 是不可度量的，则：

$$\bar{x}^{(t)}=\arg\max_{x_j^{(t)},1\leqslant j\leqslant N1}\sum_{i=1}^{N1}I(x_i^{(t)}=x_j^{(t)})$$

$$\hat{x}_i^{(t)}=\begin{cases}x_i^{(t)}, & i=1,2,\cdots,N1\\ \bar{x}^{(t)}, & i=N1+1,N1+2,\cdots,N\end{cases}$$

同类均值插补

采用均值插补有个缺点：含有缺失值的属性 $x^{(t)}$ 上的所有缺失值都填补为同样的值。同类均值插补的思想是：首先将样本进行分类，然后以该类中样本的均值来插补缺失值。

给定数据集 $D=\{(\vec{\boldsymbol{x}}_1,y_1),(\vec{\boldsymbol{x}}_2,y_2),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\}$，其中 $\vec{\boldsymbol{x}}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(d)})^T,i=1,2,\cdots,N$。假设 $x^{(t)}$ 属性含有缺失值。将数据集划分为 $D_l=\{(\vec{\boldsymbol{x}}_1,y_1),(\vec{\boldsymbol{x}}_2,y_2),\cdots,(\vec{\boldsymbol{x}}_l,y_l)\}$ 和 $D_u=\{(\vec{\boldsymbol{x}}_{l+1},y_{l+1}),(\vec{\boldsymbol{x}}_{l+2},y_{l+2}),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\}$，其中 $x^{(t)}$ 在 D_l 上含有有效数据，在 D_u 上有缺失值。

首先利用层次聚类算法对 D_l 进行聚类。设聚类的结果为 K 个簇 $C_1,C_2,\cdots,C_K$。计算这 K 个簇在 $x^{(t)}$ 上的均值 $\mu_1,\mu_2,\cdots,\mu_K$。

- 对于 $\vec{\boldsymbol{x}}_i\in D_l$，有 $\hat{x}_i^{(t)}=x_i^{(t)}$。
- 对于 $\vec{\boldsymbol{x}}_i\in D_u$，先对其进行聚类预测，设它被判定为属于簇 $C_k,(1\leqslant k\leqslant K)$，则有 $\hat{x}_i^{(t)}=\mu_k$。

建模预测

建模预测的思想是：将缺失的属性作为预测目标来预测。给定数据集 $D=\{(\vec{\boldsymbol{x}}_1,y_1),(\vec{\boldsymbol{x}}_2,y_2),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\}$，其中 $\vec{\boldsymbol{x}}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(d)})^T,i=1,2,\cdots,N$。假设 $x^{(t)}$ 属性含有缺失值，且假设 $(\vec{\boldsymbol{x}}_1,\vec{\boldsymbol{x}}_2,\cdots,\vec{\boldsymbol{x}}_{N1})$ 在 $x^{(t)}$ 属性上含有有效值，$(\vec{\boldsymbol{x}}_{N1+1},\vec{\boldsymbol{x}}_{N1+2},\cdots,\vec{\boldsymbol{x}}_N)$ 在 $x^{(t)}$ 属性为缺失值。

构建新的训练数据集为 $D_t=\{(\hat{\vec{\boldsymbol{x}}}_1,x_1^{(t)}),(\hat{\vec{\boldsymbol{x}}}_2,x_2^{(t)}),\cdots,(\hat{\vec{\boldsymbol{x}}}_{N1},x_{N1}^{(t)})\}$，构建待预测数据集为 $D_p=\{\hat{\vec{\boldsymbol{x}}}_{N1+1},\hat{\vec{\boldsymbol{x}}}_{N1+2},\cdots,\hat{\vec{\boldsymbol{x}}}_N\}$。其中

$$\hat{\vec{\boldsymbol{x}}}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(t-1)},x_i^{(t+1)},\cdots,x_i^{(d)})^T,i=1,2,\cdots,N$$

利用现有的机器学习算法从 D_t 中学习，设学到的算法为 f，则：

$$\hat{x}_i^{(t)}=\begin{cases}x_i^{(t)}, & i=1,2,\cdots,N1\\ f(\hat{\vec{\boldsymbol{x}}}_i), & i=N1+1,N1+2,\cdots,N\end{cases}$$

这种方法的效果较好，但是该方法有个根本的缺陷：如果其他属性和缺失属性 $x^{(t)}$ 无关，则预测的结果毫无意义。但是如果预测结果相当准确，则说明这个缺失属性 $x^{(t)}$ 是没必要考虑纳入数据集中的。一般的情况是介于两者之间。

高维映射

高维映射的思想是：将属性映射到高维空间。它采用了后面小节介绍的独热码编码的技术。给定数据集 $D=\{(\vec{\boldsymbol{x}}_1,y_1),(\vec{\boldsymbol{x}}_2,y_2),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\}$，其中 $\vec{\boldsymbol{x}}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(d)})^T,i=1,2,\cdots,N$。假设 $x^{(t)}$ 属性含有缺失值，$x^{(t)}$ 属性的取值为离散值 $\{a_{t,1},a_{t,2}\cdots,a_{t,K}\}$，一共 K 个取值。将该属性扩展为 $K+1$ 个属性 $(x^{(t,1)},x^{(t,2)}\cdots,x^{(t,K)},x^{(t,K+1)})$，其中：

- 若 $x^{(t)} = a_{t,j}, j = 1,2,\cdots,K$ ，则 $x^{(t,j)} = 1$；
- 若 $x^{(t)}$ 属性值缺失，则 $x^{(t,K+1)} = 1$；
- 其他情况下 $x^{(t,j)} = 0$。

于是有：

$$\hat{\vec{x}}_i = (x_i^{(1)}, x_i^{(2)}, \cdots, x_i^{(t-1)}, x_i^{(t,1)}, x_i^{(t,2)} \cdots, x_i^{(t,K)}, x_i^{(t,K+1)}, x_i^{(t+1)}, \cdots, x_i^{(d)})^T \quad ,i = 1,2,\cdots,N$$

$$x_i^{(t,j)} = \begin{cases} 1, & \text{if} \quad j = K+1 \text{ and } x_i^{(t)} \text{ miss} \\ 1, & \text{if} \quad 1 \leqslant j \leqslant K \text{ and } x_i^{(t)} = a_{t,j} \\ 0, & \text{else} \end{cases} \quad ,j = 1,2,\cdots,K+1$$

这种做法是最精确的做法，它完全保留了所有的信息，也未增加任何额外的信息。比如Google、百度的CTR预估模型，预处理时会把所有变量都这样处理，达到几亿维。这样做的好处是完整保留了原始数据的全部信息、不用考虑缺失值。但它的缺点也很明显，就是计算量大大提升。而且只有在样本量非常大的时候效果才好，否则会因为过于稀疏，效果很差。

多重插补

多重插补（Multiple Imputation:MI）认为待插补的值是随机的，它的值来自于已观测到的值。具体实践上通常是估计出待插补的值，然后再加上不同的噪声，形成多组可选插补值。根据某种选择依据，选取最合适的插补值。

多重插补方法分为三个步骤：

1. 通过变量之间的关系对缺失数据进行预测，利用蒙特卡洛方法生成多个完整的数据集；
2. 在每个完整的数据集上进行训练，得到训练后的模型以及评价函数值；
3. 对来自各个完整的数据集的结果，根据评价函数值进行选择，选择评价函数值最大的模型，其对应的插值就是最终的插补值。

压缩感知及矩阵补全

（1）压缩感知 考虑信号补全问题。假定有长度为 N 的离散信号 $\vec{x}$。根据奈奎斯特采样定理，当采样频率达到 $\vec{x}$ 最高频率的两倍时，采样后的信号就保留了原信号的全部信息。

假定以远小于奈奎斯特采样定理要求的采样频率进行采样，获得采样信号 $\vec{y}$，其长度为 M。其中 $M \ll N$。则有：

$$\vec{y} = \Phi\vec{x}$$

其中 $\Phi \in \mathbb{R}^{M\times N}$ 是对信号 $\vec{x}$ 的测量矩阵，它确定了采样的方式。

若给定测量值 $\vec{\boldsymbol{y}}$ 、测量矩阵 $\boldsymbol{\Phi}$，则还原出原始信号 $\vec{\boldsymbol{x}}$ 非常困难。因为当 $M \ll N$ 时，$\vec{\boldsymbol{y}} = \boldsymbol{\Phi}\vec{\boldsymbol{x}}$ 是一个欠定方程，无法简单地求出数值解。

假设存在某种线性变换 $\boldsymbol{\Psi} \in \mathbb{R}^{N\times N}$ ，使得 $\vec{\boldsymbol{x}} = \boldsymbol{\Psi}\vec{\boldsymbol{s}}$，其中 $\vec{\boldsymbol{s}}$ 也和 $\vec{\boldsymbol{x}}$ 一样是 N 维列向量，则 $\vec{\boldsymbol{y}} = \boldsymbol{\Phi}\vec{\boldsymbol{x}} = \boldsymbol{\Phi}\boldsymbol{\Psi}\vec{\boldsymbol{s}}$。令 $\boldsymbol{A} = \boldsymbol{\Phi}\boldsymbol{\Psi} \in \mathbb{R}^{M\times N}$ ，则 $\vec{\boldsymbol{y}} = \boldsymbol{A}\vec{\boldsymbol{s}}$ 。若能从 $\vec{\boldsymbol{y}}$ 中恢复 $\vec{\boldsymbol{s}}$，则能通过 $\vec{\boldsymbol{x}} = \boldsymbol{\Psi}\vec{\boldsymbol{s}}$ 从 $\vec{\boldsymbol{y}}$ 中恢复出 $\vec{\boldsymbol{x}}$。

虽然从数学上看没有什么意义，但是在实际应用中发现：若 $\vec{\boldsymbol{s}}$ 具有稀疏性（即大量的分量为零），则该问题能够很好地求解。这就是压缩感知的基本思想。

压缩感知通过利用信号本身所具有的稀疏性，从部分观测样本中恢复原信号。压缩感知分为感知测量和重构恢复两个阶段。

- 感知测量：此阶段对原始信号进行处理以获得稀疏样本表示。常用的手段是傅里叶变换、小波变换、字典学习、稀疏编码等。
- 重构恢复：此阶段基于稀疏性从少量观测中恢复原信号。这是压缩感知的核心。

这里介绍限定等距性（Restricted Isometry Property，RIP）：对于大小为 $M\times N, M \ll N$ 的矩阵 $\boldsymbol{A}$，若存在常数 $\delta_k \in (0,1)$，使得对于任意向量 $\vec{\boldsymbol{s}}$ 和 $\boldsymbol{A}$ 的所有子矩阵 $\boldsymbol{A}_k \in \mathbb{R}^{M\times k}$ 都有：

$$(1-\delta_k)||\vec{\boldsymbol{s}}||_2^2 \leqslant ||\boldsymbol{A}_k\vec{\boldsymbol{s}}||_2^2 \leqslant (1+\delta_k)||\vec{\boldsymbol{s}}||_2^2$$

则称 $\boldsymbol{A}$ 满足 k 限定等距性 k-RIP。此时可通过求解下面的最优化问题恢复出稀疏信号 $\vec{\boldsymbol{s}}$，进而恢复出 $\vec{\boldsymbol{y}}$：

$$\min_{\vec{s}} ||\vec{\boldsymbol{s}}||_0$$
$$s.t. \quad \vec{\boldsymbol{y}} = \boldsymbol{A}\vec{\boldsymbol{s}}$$

这里 L_0 范数表示向量中非零元素的个数。该最优化问题涉及 L_0 范数最小化，这是个NP难问题。但是 L_1 范数最小化在一定条件下与 L_0 范数最小化问题共解，于是实际上只需要求解最优化问题：

$$\min_{\vec{s}} ||\vec{\boldsymbol{s}}||_1$$
$$s.t. \quad \vec{\boldsymbol{y}} = \boldsymbol{A}\vec{\boldsymbol{s}}$$

可以将该问题转化为LASSO等价形式，然后通过近端梯度下降法来求解。

（2）矩阵补全 一个现实的例子是对电影进行评分。假设有 100 部电影让网友评分，通常每个网友只是观赏过部分电影，因此他们只会对这 100 部电影的一部分进行评分。因此我们采集的仅仅是部分有效信息，其中有大量的未知项，用 ? 表示：

	电影 1	电影 2	电影 3	...	电影 100
网友 1	5	?	?	...	3
网友 2	?	7	8	...	5
...	...	...	...	...	...
网友 xx	8	6	?	...	?

矩阵补全技术基于压缩感知的思想，将由网友评价得到的数据当作部分信号，从而恢复出完整信号。

矩阵补全matrix completion技术解决的问题是：

$$\min_{\boldsymbol{X}} rank(\boldsymbol{X}) s.t. \quad x_{i,j} = a_{i,j}, (i,j) \in \Omega$$

其中

$$\boldsymbol{X} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{bmatrix} \quad \boldsymbol{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}$$

$\boldsymbol{A}$ 为观测矩阵；Ω 为 $\boldsymbol{A}$ 中所有有数值的下标的集合；$\boldsymbol{X}$ 为需要恢复的稀疏信号；$rank(\boldsymbol{X})$ 为矩阵 $\boldsymbol{X}$ 的秩。该最优化问题也是一个 NP 难问题。

考虑到 $rank(\boldsymbol{X})$ 在集合 $\{\boldsymbol{X} \in \mathbb{R}^{m \times n} : ||\boldsymbol{X}||_F^2 \leqslant 1\}$ 上的凸包是 $\boldsymbol{X}$ 的核范数nuclear norm：

$$||\boldsymbol{X}||_* = \sum_{j=1}^{\min\{m,n\}} \sigma_j(\boldsymbol{X})$$

其中 $\sigma_j(\boldsymbol{X})$ 表示 $\boldsymbol{X}$ 的奇异值，于是可以通过最小化矩阵核范数来近似求解：

$$\min_{\boldsymbol{X}} ||\boldsymbol{X}||_*$$
$$s.t. \quad x_{i,j} = a_{i,j}, (i,j) \in \Omega$$

该最优化问题是一个凸优化问题，可以通过半正定规划（Semi-Definite Programming，SDP）求解。

缺失值插补小结

插补处理只是将未知值补以我们的主观估计值，不一定完全符合客观事实。以上的分析都是理论分析，对于缺失值由于它本身无法观测，也就不可能知道它的缺失所属类型，也

就无从估计一个插补方法的插补效果。另外这些方法通用于各个领域，具有了普遍性，那么针对一个领域的专业的插补效果就不会很理想，正因为这个原因，很多专业数据分析人员通过他们对行业的理解，手动对缺失值进行插补的效果反而可能比这些方法更好。

在数据挖掘过程中为了不放弃大量的信息，采用人为干涉缺失值的情况便是缺失值的插补，但无论是哪种处理方法都会影响变量间的相互关系，在对不完备信息进行补齐处理的同时，或多或少地改变了原始数据的信息系统，对以后的分析存在潜在的影响，所以对缺失值的处理一定要慎重。

11.2.4 特征编码

特征二元化

特征二元化的过程是将数值型的属性转换为布尔值的属性。通常用于假设属性取值分布为伯努利分布的情形。

特征二元化的算法比较简单。假设数据集 $D = \{(\vec{\boldsymbol{x}}_1, y_1), (\vec{\boldsymbol{x}}_2, y_2), \cdots, (\vec{\boldsymbol{x}}_N, y_N)\}$，$\vec{\boldsymbol{x}}_i = (x_i^{(1)}, x_i^{(2)}, \cdots, x_i^{(d)})^T$。对于某个属性 $x^{(j)}$，其取值集合为 $\{x_1^{(j)}, x_2^{(j)}, \cdots, x_N^{(j)}\}$ 。若指定一个阈值 ϵ，则:

- ❑ 当 $x^{(j)} \geqslant \epsilon$ 时，该属性二元化后的值为 1，即 $\hat{x}^{(j)} = 1$；
- ❑ 当 $x^{(j)} < \epsilon$ 时，该属性二元化后的值为 0，即 $\hat{x}^{(j)} = 0$。

独热编码

对于有些属性，如[男，女]，[中国，美国，英国]等是非数值属性。可以构建一个映射，将这些非数值属性映射到整数，如[男，女]属性中，将男映射为整数 1，女映射为整数 0。该方法的优点是简单，但问题是在这种处理方式中，无序的属性被看成有序的（男和女无法比较大小，但是 1 和 0 有大小）。解决该问题的方案是采用独热码（One-Hot Encoding）。

One-Hot Encoding采用 N 位状态寄存器来对 N 个可能的取值进行编码，每个状态都由独立的寄存器位来表示，并且在任意时刻只有其中的一位有效。假设某个属性的取值为非数值的离散集合 [离散值1, 离散值2, ⋯, 离散值m]，则针对该属性的编码为一个 m 元的元组:

$$(v_1, v_2, \cdots, v_m) v_i \in \{0, 1\}, \quad i = 1, 2, \cdots, m$$

且 $(v_1, v_2, \cdots, v_m)$ 的分量有且仅有一个为 1，其余的分量均为 0。

对于性别编码：男编码为 $(1, 0)$；女编码为 $(0, 1)$。对于国家编码：中国编码为 $(1, 0, 0)$；美国编码为 $(0, 1, 0)$；英国编码为 $(0, 0, 1)$。

样本的编码为其属性的编码元组的拼接。如果某个样本为:男，中国,则其编码为 $(1, 0, 1, 0, 0)$。由此可以看出One-Hot Encoding 的三个优点：

- 能够处理非数值属性；
- 在一定程度上也扩充了特征。如性别本身是一个属性，经过One-Hot Encoding编码之后变成了男或女两个属性；
- 编码后的属性是稀疏的，存在大量的零元分量。

11.2.5　数据标准化、正则化

数据标准化

数据标准化是将样本的属性缩放到某个指定的范围。数据标准化的两个原因如下。

- 某些算法要求样本数据具有零均值和单位方差。
- 样本不同属性具有不同量级时，消除数量级的影响。如图 11.1 所示为两个属性的目标函数的等高线。
  - 数量级的差异将导致量级较大的属性占据主导地位。从图中可以看到：如果样本的某个属性的量级特别巨大，将原本为椭圆的等高线压缩成直线，从而使得目标函数值仅依赖于该属性。
  - 数量级的差异将导致迭代收敛速度减慢。从图中可以看到：原始的特征进行梯度下降时，每一步梯度的方向会偏离最小值（等高线中心点）的方向，迭代次数较多。标准化后进行梯度下降时，每一步梯度的方向都几乎指向最小值（等高线中心点）的方向，迭代次数较少。
  - 所有依赖于样本距离的算法对于数据的数量级都非常敏感。如 k 近邻算法需要计算距离当前样本最近的 k 个样本。当属性的量级不同时，选取的最近的 k 个样本也会不同。

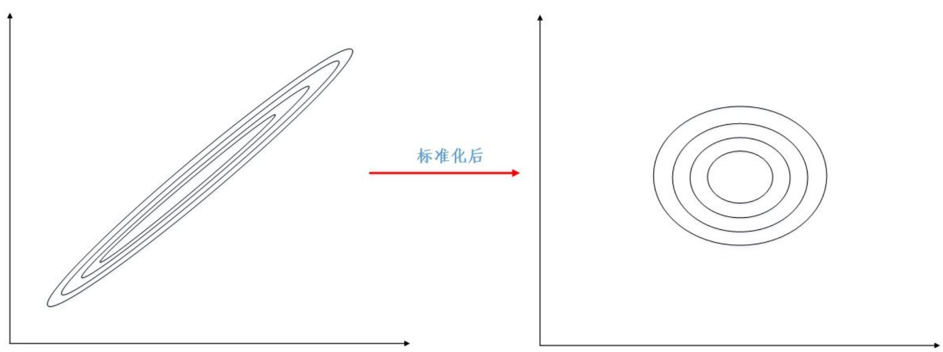


图 11.1　standardization

设数据集 $D=\{(\vec{\boldsymbol{x}}_1,y_1),(\vec{\boldsymbol{x}}_2,y_2),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\},\vec{\boldsymbol{x}}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(d)})^T$ 。常用的标准化算法有：

- min-max标准化：对于每个属性 $x^{(j)}, j=1,2,\cdots,d$，根据下式计算标准化后的属性值：

$$\hat{x}_i^{(j)}=\frac{x_i^{(j)}-\min x^{(j)}}{\max x^{(j)}-\min x^{(j)}},\quad i=1,2,\cdots,N;j=1,2,\cdots,d$$
$$\hat{\vec{\boldsymbol{x}}}_i=(\hat{x}_i^{(1)},\hat{x}_i^{(2)},\cdots,\hat{x}_i^{(d)})^T,\quad i=1,2,\cdots,N$$

其中 $\max x^{(j)}=\max\{x_1^{(j)},x_2^{(j)},\cdots,x_N^{(j)}\}$ 为属性 $x^{(j)}$ 的最大值；$\min x^{(j)}=\min\{x_1^{(j)},x_2^{(j)},\cdots,x_N^{(j)}\}$ 为属性 $x^{(j)}$ 的最小值。标准化之后，样本 $\vec{\boldsymbol{x}}_i$ 的所有属性值都在 [0,1] 之间。

- z-score标准化：对于每个属性 $x^{(j)}, j=1,2,\cdots,d$，先计算该属性的标准值和标准：

$$\mu^{(j)}=\frac{1}{N}\sum_{i=1}^{N}x_i^{(j)}$$
$$\sigma^{(j)}=\sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i^{(j)}-\mu^{(j)})^2}$$

然后根据下式计算标准化后的属性值：

$$\hat{x}_i^{(j)}=\frac{x_i^{(j)}-\mu^{(j)}}{\sigma^{(j)}},\quad i=1,2,\cdots,N;j=1,2,\cdots,d$$
$$\hat{\vec{\boldsymbol{x}}}_i=(\hat{x}_i^{(1)},\hat{x}_i^{(2)},\cdots,\hat{x}_i^{(d)})^T,\quad i=1,2,\cdots,N$$

标准化之后，样本集的所有属性的均值都为 0，标准差均为 1。

> 均值和标准差都是在样本集上定义的，而不是在单个样本上定义的。

数据正则化

数据正则化是将样本的某个范数（如 L_1 范数）缩放到单位 1。正则化的过程是针对单个样本的，对于每个样本将样本缩放到单位范数。通常如果使用二次型（如点积）或者其他核方法计算两个样本之间的相似性时，该方法会很有用。

> 标准化是针对某个属性的（需要用到所有样本在该属性上的值）。

设数据集 $D=\{(\vec{\boldsymbol{x}}_1,y_1),(\vec{\boldsymbol{x}}_2,y_2),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\},\vec{\boldsymbol{x}}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(d)})^T$。对于样本 $\vec{\boldsymbol{x}}_i$，首先计算其 L_p 范数：

$$L_p(\vec{\boldsymbol{x}}_i)=(|x_i^{(1)}|^p+|x_i^{(2)}|^p+\cdots+|x_i^{(d)}|^p)^{1/p}$$

样本 $\vec{\boldsymbol{x}}_i$ 正则化后的结果为：每个属性值除以其 L_p 范数：

$$\hat{\vec{\boldsymbol{x}}}_i = (\frac{x_i^{(1)}}{L_p(\vec{\boldsymbol{x}}_i)}, \frac{x_i^{(2)}}{L_p(\vec{\boldsymbol{x}}_i)}, \cdots, \frac{x_i^{(d)}}{L_p(\vec{\boldsymbol{x}}_i)})^T$$

11.2.6 特征选择

特征选择原理

在学习任务中，当给定了属性集，其中某些属性可能对于学习来说是很关键的，但是有些属性可能就没有什么用。

- 对于当前学习任务有用的属性称为相关特征（relevant feature）。
- 对当前学习任务没有用的属性称为无关特征（irrelevant feature）。

从给定的特征集合中选出相关特征子集的过程称为特征选择（feature selection）。进行特征选择有两个重要原因：

- 首先维数灾难问题就是由于属性过多造成的。若挑选出重要特征，使得后续学习过程仅仅需要在这小部分特征上构建模型，则维数灾难问题会大大减轻。

  > 特征选择与降维技术是处理高维数据的两大主要方法。

- 其次去除不相关特征通常会降低学习任务的难度。

进行特征选择必须确保不丢失重要特征，若重要信息缺失则学习效果会大打折扣。常见的特征选择方法大致可分为三类：过滤式（filter）、包裹式（wrapper）、嵌入式（embedding）。

> 有一类特征称为冗余特征（redundant feature），它所包含的信息能从其他特征中推演出来。冗余特征在很多时候不起作用，去除它们能够减轻学习过程的负担；但如果冗余特征恰好对应了完成学习任务所需要的某个中间概念，则该冗余特征是有益的，能降低学习任务的难度。

过滤式选择

过滤式方法先对数据集进行特征选择，然后再训练学习器。特征选择过程与后续学习器无关。Relief（Relevant Features）就是一种著名的过滤式特征选择方法。

给定训练集 $D = \{(\vec{\boldsymbol{x}}_1, y_1), (\vec{\boldsymbol{x}}_2, y_2), \cdots, (\vec{\boldsymbol{x}}_N, y_N)\}, \vec{\boldsymbol{x}}_i = (x_i^{(1)}, x_i^{(2)}, \cdots, x_i^{(d)})^T, y_i \in \{-1, +1\}$。Relief的步骤如下。

- 对于每个样本 $\vec{\mathbf{x}}_i, i=1,2,\cdots,N$：
  - 先在 $\vec{\mathbf{x}}_i$ 同类样本中寻找其最近邻 $\vec{\mathbf{x}}_{i,nh}$，称为猜中近邻near-hit。
  - 然后从 $\vec{\mathbf{x}}_i$ 的异类样本中寻找其最近邻 $\vec{\mathbf{x}}_{i,nm}$，称为猜错近邻near-miss。
  - 再然后计算 $\vec{\delta}_i=(\delta_i^{(1)},\delta_i^{(2)},\cdots,\delta_i^{(d)})^T$ 的对应于属性 j 的分量 $(j=1,2,\cdots,d)$。

$$\delta_i^{(j)}=\sum_{i=1}^{N}\left(-\text{diff}(x_i^{(j)},x_{i,nh}^{(j)})^2+\text{diff}(x_i^{(j)},x_{i,nm}^{(j)})^2\right)$$

其中 $\text{diff}(x_a^{(j)},x_b^{(j)})$ 为两个样本在属性 j 上的差异值，其结果取决于该属性是离散的还是连续的：

- 如果属性 j 是离散的，则

$$\text{diff}(x_a^{(j)},x_b^{(j)})=\begin{cases}0, & \text{if}\quad x_a^{(j)}=x_b^{(j)}\\ 1, & \text{else}\end{cases}$$

- 如果属性 j 是连续的，则

$$\text{diff}(x_a^{(j)},x_b^{(j)})=|x_a^{(j)}-x_b^{(j)}|$$

> 此时 $x_a^{(j)},x_b^{(j)}$ 已经标准化到 [0,1] 区间。

  - 计算 $\vec{\delta}$ 为 $\vec{\delta}_i$ 的均值

$$\vec{\delta}=\sum_{i=1}^{N}\frac{1}{N}\vec{\delta}_i$$

  - 根据指定的阈值 τ，如果 $\delta^{(j)}>\tau$，则样本属性 j 被选中。

> 也可以指定要选取的特征个数 k，此时 $\vec{\delta}$ 的分量最大的 k 个特征被选中。

Relief 是为二分类问题设计的，其推广形式 Relief-F 用于处理多分类问题。假定数据集 D 中的样本类别为 $c_1,c_2,\cdots,c_K$。对于样本 $\vec{\mathbf{x}}_i$，假设其类别为 $y_i=c_k$。Relief-F 与Relief 的区别如下。

- Relief-F先在类别 c_k 的样本中寻找 $\vec{\mathbf{x}}_i$ 的最近邻 $\vec{\mathbf{x}}_{i,nh}$ 作为猜中近邻。
- 然后在 c_k 之外的每个类别中分别找到一个 $\vec{\mathbf{x}}_i$ 的最近邻 $\vec{\mathbf{x}}_{i,mn,l}, l=1,2,\cdots,K; l\neq k$ 作为猜错近邻。

- 计算 $\vec{\delta}_i = (\delta_i^{(1)}, \delta_i^{(2)}, \cdots, \delta_i^{(d)})^T$ 的对应于属性 j 的分量为 $(j = 1, 2, \cdots, d)$

$$\delta_i^{(j)} = \sum_{i=1}^{N} \left(-\text{diff}(x_i^{(j)}, x_{i,nh}^{(j)})^2 + \sum_{l \neq k} \left(p_l \times \text{diff}(x_i^{(j)}, x_{i,nm,l}^{(j)})^2 \right) \right)$$

  其中 p_l 为第 l 类的样本在数据集 D 中所占的比例。

包裹式选择

包裹式特征选择直接把最终将要使用的学习器的性能作为特征子集的评价准则。其优点是：由于包裹式特征选择方法直接针对特定学习器进行优化，因此通常包裹式特征选择比过滤式特征选择更好。其缺点是：由于特征选择过程中需要多次训练学习器，因此计算开销通常比过滤式特征选择要大得多。

`LVW`（`Las Vegas Wrapper`）是一个典型的包裹式特征选择方法。它在`Las Vegas method`框架下使用随机策略来进行子集搜索，并以最终分类器的误差作为特征子集的评价标准。

`LVW`算法

- 输入：
  - 数据集 $D = \{(\vec{x}_1, y_1), (\vec{x}_2, y_2), \cdots, (\vec{x}_N, y_N)\}, \vec{x}_i = (x_i^{(1)}, x_i^{(2)}, \cdots, x_i^{(d)})^T$；
  - 特征集 $A = \{x^{(1)}, x^{(2)}, \cdots, x^{(d)}\}$；
  - 学习器 `estimator`；
  - 迭代停止条件 T。
- 输出：最优特征子集 A^*。
- 算法步骤如下。
  - 初始化：将候选的最优特征子集 $\tilde{A}^* = A$，然后学习器 `estimator`在特征子集 $\tilde{A}^*$ 上使用交叉验证法进行学习，通过学习结果评估学习器 `estimator`的误差 err^*。
  - 迭代，停止条件为迭代次数到达 T 。迭代过程为：
    - 随机产生特征子集 A'；
    - 学习器 `estimator`在特征子集 A' 上使用交叉验证法进行学习，通过学习结果评估学习器 `estimator`的误差 err；
    - 如果 err 比 err^* 更小，或者 $err = err^*$，但是 A' 的特征数量比 $\tilde{A}^*$ 的特征数量更少，则将 A 作为候选的最优特征子集：

$$\tilde{A}^* = A'; \quad err^* = err$$

  - 最终 $A^* = \tilde{A}^*$。

注意：若初始特征数量很多、T 设置较大、以及每一轮训练的时间较长，则很可能算法运行很长时间都不会停止。

嵌入式选择和 L_1 正则化

前两种特征选择方法中，特征选择过程和学习器训练过程有明显的分别。而嵌入式特征选择是在学习器训练过程中自动进行了特征选择。

以最简单的线性回归模型为例。给定数据集 $\boldsymbol{D}=\{(\vec{\boldsymbol{x}}_1,y_1),(\vec{\boldsymbol{x}}_2,y_2),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\},\vec{\boldsymbol{x}}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(d)})^T,y_i\in\mathbb{R}$，若损失函数为平方损失函数，则优化目标为：

$$\min_{\vec{\boldsymbol{w}}}\sum_{i=1}^N(y_i-\vec{\boldsymbol{w}}^T\vec{\boldsymbol{x}}_i)^2$$

引入正则化项。

- 如果使用 L_2 范数正则化，则优化目标为：

$$\min_{\vec{\boldsymbol{w}}}\sum_{i=1}^N(y_i-\vec{\boldsymbol{w}}^T\vec{\boldsymbol{x}}_i)^2+\lambda||\vec{\boldsymbol{w}}||_2^2,\quad \lambda>0$$

  此时称为岭回归（ridge regression）。

- 如果使用 L_1 范数正则化，则优化目标为：

$$\min_{\vec{\boldsymbol{w}}}\sum_{i=1}^N(y_i-\vec{\boldsymbol{w}}^T\vec{\boldsymbol{x}}_i)^2+\lambda||\vec{\boldsymbol{w}}||_1,\quad \lambda>0$$

  此时称为LASSO（Least Absolute Shrinkage and Selection Operator）回归。

引入 L_1 范数除了降低过拟合风险之外，还有一个好处：它求得的 $\vec{\boldsymbol{w}}$ 会有较多的分量为零。即它更容易获得稀疏sparse解。

假设 $\vec{\boldsymbol{w}}=(w^{(1)},w^{(2)},\cdots,w^{(d)})^T$ 的解为 $(w^{(1*)},w^{(2*)},\cdots,w^{(v*)},0,0,\cdots,0)^T$，即前 v 个分量非零，后面的 $d-v$ 个分量为零；则这意味着初始的 d 个特征中，只有前 v 个特征才会出现在最终模型中。

于是基于 L_1 正则化的学习方法就是一种嵌入式特征选择方法，其特征选择过程也就是学习器训练过程。

L_1 正则化问题的求解可以用近端梯度下降（Proximal Gradient Descent，PGD）算法求解。

11.2.7 稀疏表示和字典学习

对于 $\boldsymbol{D}=\{(\vec{\boldsymbol{x}}_1,y_1),(\vec{\boldsymbol{x}}_2,y_2),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\},\vec{\boldsymbol{x}}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(d)})^T,y_i\in\mathbb{R}$。构建矩阵 $\boldsymbol{D}=(\vec{\boldsymbol{x}}_1,\vec{\boldsymbol{x}}_2,\cdots,\vec{\boldsymbol{x}}_N)^T$，其内容为：

$$\boldsymbol{D}=\begin{bmatrix} x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(d)} \\ x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(d)} \\ \vdots & \vdots & \ddots & \vdots \\ x_N^{(1)} & x_N^{(2)} & \cdots & x_N^{(d)} \end{bmatrix}$$

其中每一行对应一个样本，每一列对应一个特征。考虑以下两类情况：

- 矩阵中可能许多列与当前学习任务无关。如果通过特征选择去除这些列，则学习器训练过程仅需要在较小的矩阵上进行。这就是特征选择要解决的问题。
- $\boldsymbol{D}$ 中有大量元素为 0 ，这称为稀疏矩阵。如果数据集具有高度的稀疏性，则该问题很可能是线性可分的。而对于线性支持向量机，能取得更佳的性能。另外稀疏矩阵有很多很高效的存储方法，可以节省存储空间。这就是字典学习要考虑的问题。

字典学习（`dictionary learning`）：学习一个字典，通过该字典将样本转化为合适的稀疏表示形式。稀疏编码（`sparse coding`）：获取样本的稀疏表达，不一定需要通过字典。这两者通常是在同一个优化求解过程中完成的。因此这里不做区分，统称为字典学习。

给定数据集 $D=\{(\vec{\boldsymbol{x}}_1,y_1),(\vec{\boldsymbol{x}}_2,y_2),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\},\vec{\boldsymbol{x}}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(d)})^T$ 。希望对样本 $\vec{\boldsymbol{x}}_i$，学习到它的一个稀疏表示 $\vec{\alpha}_i\in\mathbb{R}^k$ (一个 k 维列向量)。一个自然的想法是进行线性变换，即寻找一个矩阵 $\boldsymbol{P}\in\mathbb{R}^{k\times d}$，使得 $\boldsymbol{P}\vec{\boldsymbol{x}}_i=\vec{\alpha}_i$。

现在的问题是：既不知道变换矩阵 $\boldsymbol{P}$ ，也不知道 $\vec{\boldsymbol{x}}_i$ 的稀疏表示 $\vec{\alpha}_i$。因此要求解它们，求解的目标是：

- 根据 $\vec{\alpha}_i$ 能正确还原 $\vec{\boldsymbol{x}}_i$，或者还原的误差最小。
- $\vec{\alpha}_i$ 尽量稀疏，即它的分量尽量为零。

> 我们求解的不是变换矩阵 $\boldsymbol{P}$ ，而是逆向的变换矩阵 $\boldsymbol{B}$。

因此给出字典学习的最优化目标：

$$\min_{\boldsymbol{B},\vec{\alpha}_i}\sum_{i=1}^N||\vec{\boldsymbol{x}}_i-\boldsymbol{B}\vec{\alpha}_i||_2^2+\lambda\sum_{i=1}^N||\vec{\alpha}_i||_1$$

其中 $\boldsymbol{B}\in\mathbb{R}^{d\times k}$ 为字典矩阵，k 称为字典的词汇量，通常由用户指定。上式中第一项希望 $\vec{\alpha}_i$ 能够很好地重构 $\vec{\boldsymbol{x}}_i$，第二项则希望 $\vec{\alpha}_i$ 尽可能地稀疏（即尽可能多的项为 0）。

求解该问题采用类似LASSO的解法，但是使用变量交替优化的策略：

- 第一步：固定字典 $\boldsymbol{B}$，为每一个样本 $\vec{\boldsymbol{x}}_i$ 找到相应的 $\vec{\alpha}_i$ 。这是通过求解下式来实现的。

$$\min_{\vec{\alpha}_i} ||\vec{\boldsymbol{x}}_i - \boldsymbol{B}\vec{\alpha}_i||_2^2 + \lambda \sum_{i=1}^{N} ||\vec{\alpha}_i||_1$$

- 第二步：根据下式，以 $\vec{\alpha}_i$ 为初值来更新字典 $\boldsymbol{B}$，即求解下式

$$\min_{\boldsymbol{B}} ||\boldsymbol{X} - \boldsymbol{B}\boldsymbol{A}||_F^2$$

其中 $\boldsymbol{X} = (\vec{\boldsymbol{x}}_1, \vec{\boldsymbol{x}}_2, \cdots, \vec{\boldsymbol{x}}_N) \in \mathbb{R}^{d\times N}$，$\boldsymbol{A} = (\vec{\alpha}_1, \vec{\alpha}_2, \cdots, \vec{\alpha}_N) \in \mathbb{R}^{k\times N}$。写成矩阵的形式为：

$$\boldsymbol{X} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_N^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_N^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(d)} & x_2^{(d)} & \cdots & x_N^{(d)} \end{bmatrix} \quad \boldsymbol{A} = \begin{bmatrix} \alpha_1^{(1)} & \alpha_2^{(1)} & \cdots & \alpha_N^{(1)} \\ \alpha_1^{(2)} & \alpha_2^{(2)} & \cdots & \alpha_N^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_1^{(k)} & \alpha_2^{(k)} & \cdots & \alpha_N^{(k)} \end{bmatrix}$$

这里 $||\cdot||_F$ 为矩阵的 Frobenius范数（所有元素的平方和的平方根）。对于矩阵 $\boldsymbol{M}$，有 $||\boldsymbol{M}||_F = \sqrt{\sum_i \sum_j |m_{ij}|^2}$。

- 反复迭代上述两步，最终即可求得字典 $\boldsymbol{B}$ 和样本 $\vec{\boldsymbol{x}}_i$ 的稀疏表示 $\vec{\alpha}^{(i)}$。其中用户可以通过设置词汇量 k 的大小来控制字典的规模，从而影响到稀疏程度。

这里有个最优化问题：

$$\min_{\boldsymbol{B}} ||\boldsymbol{X} - \boldsymbol{B}\boldsymbol{A}||_F^2$$

该问题有多种求解方法，常用的有基于逐列更新策略的KSVD算法。令 $\vec{\boldsymbol{b}}_i$ 为字典矩阵 $\boldsymbol{B}$ 的第 i 列，$\vec{\boldsymbol{a}}^i$ 表示稀疏矩阵 $\boldsymbol{A}$ 的第 i 行，则上式可以重写为：

$$\min_{\vec{\boldsymbol{b}}_i} ||\boldsymbol{X} - \sum_{j=1}^{k} \vec{\boldsymbol{b}}_j \vec{\boldsymbol{a}}^j||_F^2 = \min_{\vec{\boldsymbol{b}}_i} ||(\boldsymbol{X} - \sum_{j=1, j\neq i}^{k} \vec{\boldsymbol{b}}_j \vec{\boldsymbol{a}}^j) - \vec{\boldsymbol{b}}_i \vec{\boldsymbol{a}}^i||_F^2$$

考虑到更新字典的第 i 列 $\vec{\boldsymbol{b}}_i$ 时，其他各列都是固定的，因此令：

$$\boldsymbol{E}_i = \boldsymbol{X} - \sum_{j=1, j\neq i}^{k} \vec{\boldsymbol{b}}_j \vec{\boldsymbol{a}}^j$$

$\boldsymbol{E}_i$ 是固定的，它表示去掉 $\vec{\boldsymbol{x}}_i$ 的稀疏表示之后，稀疏表示与原样本集的误差矩阵。那么最优化问题转换为：

$$\min_{\vec{\boldsymbol{b}}_i} ||\boldsymbol{E}_i - \vec{\boldsymbol{b}}_i \vec{\boldsymbol{a}}^i||_F^2$$

求解该最优化问题只需要对 $\boldsymbol{E}_i$ 进行奇异值分解，以取得最大奇异值所对应的正交向量。

然而直接对 $\boldsymbol{E}_i$ 进行奇异值分解会同时修改 $\vec{\boldsymbol{b}}_i$ 和 $\vec{\boldsymbol{a}}^i$，从而可能破坏 $\boldsymbol{A}$ 的稀疏性。因为第二步“我们以 $\vec{\boldsymbol{a}}^i$ 为初值来更新字典 $\boldsymbol{B}$”，在更新 $\boldsymbol{B}$ 前后 $\vec{\boldsymbol{a}}^i$ 的非零元所处的位置和非零元素的值很可能不一致。为避免发生这样的情况，KSVD 对 $\boldsymbol{E}_i$ 和 $\vec{\boldsymbol{a}}^i$ 进行了如下的专门处理：

- ❑ $\vec{\boldsymbol{a}}^i$ 仅保留非零元素；
- ❑ $\boldsymbol{E}_i$ 仅保留 $\vec{\boldsymbol{b}}_i$ 和 $\vec{\boldsymbol{a}}^i$ 的非零元素的乘积项，然后再进行奇异值分解，这样就保持了第一步得到的稀疏性。

11.3 Python 实践

11.3.1 二元化

scikit-learn提供的Binarizer能将数据二元化，其原型为：

```
class sklearn.preprocessing.Binarizer(threshold=0.0, copy=True)
```

参数如下。

- ❑ threshold：一个浮点数，它指定了属性阈值。低于此阈值的属性转换为 0，高于此阈值的属性转换为 1。

> 对于稀疏矩阵，该参数必须大于等于 0。

- ❑ copy：一个布尔值，如果为True，则执行原地修改（节省空间，但是修改了原始数据）。

方法如下。

- ❑ fit(X[, y])：不做任何事情，主要用于流水线Pipeline。
- ❑ transform(X[, y, copy])：将每一个样本的属性二元化。
- ❑ fit_transform(X[, y])：将每一个样本的属性二元化。

给出示例：

```
from sklearn.preprocessing import Binarizer
X=[   [1,2,3,4,5],
      [5,4,3,2,1],
      [3,3,3,3,3,],
      [1,1,1,1,1] ]
print("before transform:",X)
binarizer=Binarizer(threshold=2.5)
print("after transform:",binarizer.transform(X))
```

这里将阈值设定为 2.5。运行结果如下：

```
before transform:[[1, 2, 3, 4, 5],[5, 4, 3, 2, 1],[3, 3, 3, 3, 3], [1, 1, 1, 1, 1]]
after transform:
[[0 0 1 1 1]
 [1 1 1 0 0]
 [1 1 1 1 1]
 [0 0 0 0 0]]
```

可以看到二元化后: 所有小于 2.5 的属性的值都转换为 0；所有大于 2.5 的属性的值都转换为 1。

11.3.2 独热码

scikit-learn提供的OneHotEncoder 实现了独热码，其原型为：

```
class sklearn.preprocessing.OneHotEncoder(n_values='auto',
categorical_features='all', dtype=<class 'float'>,
sparse=True, handle_unknown='error')
```

参数

- n_values：字符串'auto'，或者一个整数，或者一个整数的数组，它指定了每个属性取值的上界。

  > 属性的取值尽量处理为从 0 开始的整数。

  - 'auto'：自动从训练数据中推断属性值取值的上界。
  - 一个整数：指定了所有属性取值的上界。
  - 一个整数的数组：每个元素依次指定了一个属性取值的上界。
- categorical_features：字符串'all'，或者下标的数组，或者是一个mask，指定哪些属性需要编码独热码。
  - 'all'：所有的属性都将编码为独热码。
  - 一个下标的数组：指定下标的属性将编码为独热码。
  - 一个mask：对应为True的属性将编码为独热码。
- dtype：一个类型，指定了独热码编码的数值类型，默认为np.float。
- sparse：一个布尔值，指定结果是否稀疏。
- handle_unknown：一个字符，如果进行数据转换时，遇到了某个集合类型的属性，但是该属性未列入categorical_features时的情形，可以指定为如下。
  - 'error'：抛出异常。
  - 'ignore'：忽略。

属性

- ❑ active_features_：一个数组，给出了激活特征。其元素意义为：如果原始数据的某个属性的某个取值在转换后数据的第 i 个属性中激活，则 i 是数组的元素。

> 该属性仅当n_values='auto'时有效。

- ❑ feature_indices_：一个数组，其元素的意义为：原始数据的第 i 个属性对应转换后数据的 [feature_indices_[i],feature_indices_[i+1]) 之间的属性。
- ❑ n_values_：一个数组，存放每个属性取值的种类（一般为训练数据中该属性取值的最大值加 1，这是因为默认每个属性取值从另开始）。

方法

- ❑ fit(X[, y])：训练OneHotEncoder。
- ❑ transform(X)：对原始数据执行独热码编码，最终结果值由选取active_features_下标中的元素组成。
- ❑ fit_transform(X[, y])：训练OneHotEncoder，然后对原始数据执行独热码编码。

给出示例：

```
from sklearn.preprocessing import OneHotEncoder
X=[   [1,2,3,4,5],
      [5,4,3,2,1],
      [3,3,3,3,3,],
      [1,1,1,1,1] ]
print("before transform:",X)
encoder=OneHotEncoder(sparse=False)
encoder.fit(X)
print("active_features_:",encoder.active_features_)
print("feature_indices_:",encoder.feature_indices_)
print("n_values_:",encoder.n_values_)
print("after transform:",encoder.transform( [[1,2,3,4,5]]))
```

这里将sparse=False（sparse=True，则每个样本的独热码为一个稀疏矩阵）。

- ❑ 第一个原始特征最大值为 5，因此第一个原始特征取值种类为 6 种（0,1,2,3,4,5），则原始数据用一个六元元组来编码：
  - ❍ 0编码为 (1,0,0,0,0,0)
  - ❍ 1编码为 (0,1,0,0,0,0)
  - ❍ 2编码为 (0,0,1,0,0,0)
  - ❍ 3编码为 (0,0,0,1,0,0)
  - ❍ 4编码为 (0,0,0,0,1,0)
  - ❍ 5编码为 (0,0,0,0,0,1)

- 第二个原始特征最大值为 4，因此第二个原始特征取值种类为 5 种（0,1,2,3,4），则原始数据用一个五元元组来编码：
  - 0编码为 (1,0,0,0,0)
  - 1编码为 (0,1,0,0,0)
  - 2编码为 (0,0,1,0,0)
  - 3编码为 (0,0,0,1,0)
  - 4编码为 (0,0,0,0,1)
- 第三个原始特征最大值为 3，因此第三个原始特征取值种类为 4 种（0,1,2,3），则原始数据用一个四元元组来编码：
  - 0编码为 (1,0,0,0)
  - 1编码为 (0,1,0,0)
  - 2编码为 (0,0,1,0)
  - 3编码为 (0,0,0,1)
- 第四个原始特征最大值为 4，因此第四个原始特征取值种类为 5 种（0,1,2,3,4），则原始数据用一个五元元组来编码：
  - 0编码为 (1,0,0,0,0)
  - 1编码为 (0,1,0,0,0)
  - 2编码为 (0,0,1,0,0)
  - 3编码为 (0,0,0,1,0)
  - 4编码为 (0,0,0,0,1)
- 第五个原始特征最大值为 5，因此第五个原始特征取值种类为 6 种（0,1,2,3,4,5），则原始数据用一个六元元组来编码：
  - 0编码为 (1,0,0,0,0,0)
  - 1编码为 (0,1,0,0,0,0)
  - 2编码为 (0,0,1,0,0,0)
  - 3编码为 (0,0,0,1,0,0)
  - 4编码为 (0,0,0,0,1,0)
  - 5编码为 (0,0,0,0,0,1)

每个样本（5 个特征）经过独热码编码后转换成 26 个特征（6+5+4+5+6=26），如图 11.2 所示。

代码运行结果如下：

```
before transform: [[1, 2, 3, 4, 5],[5, 4, 3, 2, 1],[3, 3, 3, 3, 3],[1, 1, 1, 1, 1]]
active_features_: [ 1  3  5  7  8  9 10 12 14 16 17 18 19 21 23 25]
feature_indices_: [ 0  6 11 15 20 26]
n_values_: [6 5 4 5 6]
after transform: [[ 1.  0.  0.  0.  1.  0.  0.  0.  1.  0.  0.  0.  1.  0.  0.  1.]]
```

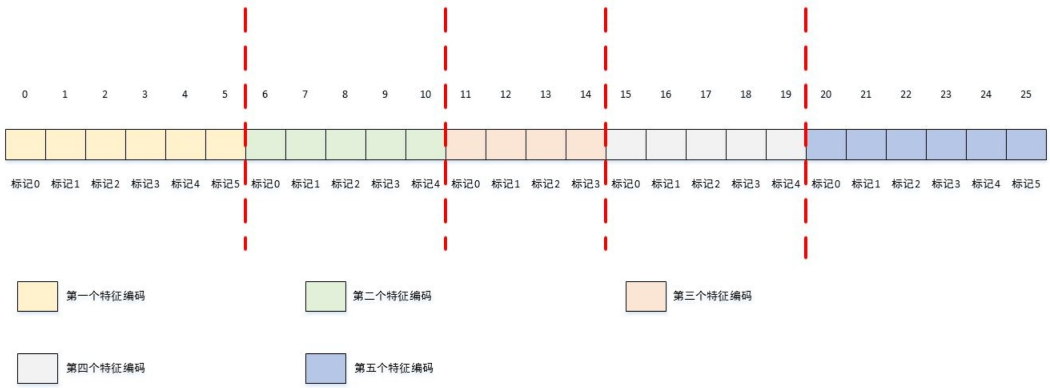


图 11.2 one_hot_encoding

- encoder.active_features_给出了激活特征。可以看到：
  - 第一个原始特征没有出现数值 0,2,4，因此转换后的 26 个特征中，对应编码的特征未激活（转换后的特征下标为 0,2,4）;
  - 第二个原始特征没有出现数值 0，因此转换后的 26 个特征中，对应编码的特征未激活（转换后的特征下标为 6）;
  - 第三个原始特征没有出现数值 0,2，因此转换后的 26 个特征中，对应编码的特征未激活（转换后的特征下标为 11,13）;
  - 第四个原始特征没有出现数值 0，因此转换后的 26 个特征中，对应编码的特征未激活（转换后的特征下标为 15）;
  - 第五个原始特征没有出现数值 0,2,4，因此转换后的 26 个特征中，对应编码的特征未激活（转换后的特征下标为 20,22,24）;

  因此激活特征数组为 [1 3 5 7 8 9 10 12 14 16 17 18 19 21 23 25]。

> 未激活特征在所有样本上的取值为 0。

- encoder.feature_indices_给出了每个原始特征在转换后特征的起始区间：
  - 第一个原始特征在转换后特征的区间 [0,6);
  - 第二个原始特征在转换后特征的区间 [6,11);
  - 第三个原始特征在转换后特征的区间 [11,15);
  - 第四个原始特征在转换后特征的区间 [15,20);
  - 第五个原始特征在转换后特征的区间 [20,26)。
- n_values_给出了每个原始属性的取值的种类，这里为 6,5,4,5,6。
- 样本[1,2,3,4,5]经过独热编码之后的值为 [[1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 1.]]。这里本应该为 26 位编码。但是考虑到剔除未激活特征，只给出了激活特征，因此这里只有 16 位编码。

11.3.3 标准化

MinMaxScaler

scikit-learn提供的MinMaxScaler实现了min-max标准化，其原型为：

```
class sklearn.preprocessing.MinMaxScaler(feature_range=(0, 1), copy=True)
```

参数

- feature_range：一个元组(min,max)，指定了预期变换之后属性的取值范围。
- copy：一个布尔值，如果为True，则执行原地修改（节省空间，但是修改了原始数据）。

属性

- min_：一个数组，给出了每个属性的原始的最小值的调整值

$$-\frac{\min x^{(j)}}{\max x^{(j)} - \min x^{(j)}}, j = 1, 2, \cdots, d$$

- scale_：一个数组，给出了每个属性的缩放倍数。
- data_min_：一个数组，给出了每个属性的原始的最小值。
- data_max_：一个数组，给出了每个属性的原始的最大值。
- data_range_：一个数组，给出了每个属性的原始的范围（最大值减最小值）。

方法

- fit(X[, y])：计算每个属性的最小值和最大值，从而为后续的转换做准备。
- transform(X)：执行属性的标准化。
- fit_transform(X[, y])：计算每个属性的最小值和最大值，然后执行属性的标准化。
- inverse_transform(X)：逆标准化，还原成原始数据。
- partial_fit(X[, y])：批量学习，学习部分数据。

给出示例：

```
from sklearn.preprocessing import MinMaxScaler
X=[   [1,5,1,2,10],
      [2,6,3,2,7],
      [3,7,5,6,4,],
      [4,8,7,8,1] ]
print("before transform:",X)
scaler=MinMaxScaler(feature_range=(0,2))
scaler.fit(X)
print("min_ is :",scaler.min_)
print("scale_ is :",scaler.scale_)
print("data_max_ is :",scaler.data_max_)
```

```
print("data_min_ is :",scaler.data_min_)
print("data_range_ is :",scaler.data_range_)
print("after transform:",scaler.transform(X))
```

这里将每个属性的值都缩放到区间[0,2]。运行结果如下：

```
before transform:[[1, 5, 1, 2, 10],[2, 6, 3, 2, 7],[3, 7, 5, 6, 4], [4, 8, 7, 8, 1]]
min_ is : [-0.66666667 -3.33333333 -0.33333333 -0.66666667 -0.22222222]
scale_ is : [ 0.66666667  0.66666667  0.33333333  0.33333333  0.22222222]
data_max_ is : [  4.   8.   7.   8.  10.]
data_min_ is : [ 1.  5.  1.  2.  1.]
data_range_ is : [ 3.  3.  6.  6.  9.]
after transform:
 [[ 0.          0.          0.          0.          2.        ]
 [ 0.66666667  0.66666667  0.66666667  0.          1.33333333]
 [ 1.33333333  1.33333333  1.33333333  1.33333333  0.66666667]
 [ 2.          2.          2.          2.          0.        ]]
```

可以看到如下。

- scaler.min_：存放的是每个属性的最小值的调整值。
- scaler.data_max_与scaler.data_min_：存放的是每个属性的最大值和最小值。
- scaler.data_range_：存放的是每个属性的最大值减去最小值。
- 标准化后，所有的属性的值都在区间[0,2]。

MaxAbsScaler

scikit-learn提供的MaxAbsScaler可以将每个属性值除以该属性的绝对值中的最大值，其原型为：

```
class sklearn.preprocessing.MaxAbsScaler(copy=True)
```

参数

- copy：一个布尔值，如果为True，则执行原地修改（节省空间，但是修改了原始数据）。

属性

- scale_：一个数组，给出了每个属性的缩放倍数的倒数。
- max_abs_：一个数组，给出了每个属性的绝对值的最大值。
- n_samples_seen_：一个整数，给出了当前已经处理的样本的数量（用于分批训练）。

方法

- fit(X[, y])：计算每个属性的绝对值的最大值，从而为后续的转换做准备。
- transform(X)：执行属性的标准化。

- fit_transform(X[, y])：计算每个属性的绝对值的最大值，然后执行属性的标准化。
- inverse_transform(X)：逆标准化，还原成原始数据。
- partial_fit(X[, y])：批量学习，学习部分数据。

给出示例：

```
from sklearn.preprocessing import MaxAbsScaler
X=[   [1,5,1,2,10],
      [2,6,3,2,7],
      [3,7,5,6,4,],
      [4,8,7,8,1] ]
print("before transform:",X)
scaler=MaxAbsScaler()
scaler.fit(X)
print("scale_ is :",scaler.scale_)
print("max_abs_ is :",scaler.max_abs_)
print("after transform:",scaler.transform(X))
```

运行结果如下：

```
before transform:[[1, 5, 1, 2, 10],[2, 6, 3, 2, 7],[3, 7, 5, 6, 4], [4, 8, 7, 8, 1]]
scale_ is : [  4.   8.   7.   8.  10.]
max_abs_ is : [  4.   8.   7.   8.  10.]
after transform:
 [[ 0.25        0.625       0.14285714  0.25        1.        ]
 [ 0.5         0.75        0.42857143  0.25        0.7       ]
 [ 0.75        0.875       0.71428571  0.75        0.4       ]
 [ 1.          1.          1.          1.          0.1       ]]
```

可以看到如下：

- scaler.scale_给出了每个属性的缩放倍数的倒数，它也是每个属性的绝对值的最大值；
- scaler.max_abs_给出了每个属性的绝对值的最大值；
- 标准化后，每个属性值的绝对值都在[0,1]之间。

StandardScaler

scikit-learn提供的StandardScaler实现了 z-score标准化。其原型为：

```
class sklearn.preprocessing.StandardScaler(copy=True, with_mean=True, with_std=True)
```

参数

- copy：一个布尔值，如果为True，则执行原地修改（节省空间，但是修改了原始数据）。

- with_mean：一个布尔值，如果为True，则缩放之前先将数据中心化（即属性值减去该属性的均值）。

> 如果元素数据是稀疏矩阵的形式，则不能指定with_mean=True。

- with_std：一个布尔值，如果为True，则缩放数据到单位方差。

属性

- scale_：一个数组，给出了每个属性的缩放倍数的倒数。
- mean_：一个数组，给出了原始数据的每个属性的均值。
- var_：一个数组，给出了原始数据的每个属性的方差。
- n_samples_seen_：一个整数，给出了当前已经处理的样本的数量（用于分批训练）。

方法

- fit(X[, y])：计算每个属性的方差和标准差，从而为后续的转换做准备。
- transform(X)：执行属性的标准化。
- fit_transform(X[, y])：计算每个属性的方差和标准差，然后执行属性的标准化。
- inverse_transform(X)：逆标准化，还原成原始数据。
- partial_fit(X[, y])：批量学习，学习部分数据。

给出示例：

```
from sklearn.preprocessing import StandardScaler
X=[   [1,5,1,2,10],
      [2,6,3,2,7],
      [3,7,5,6,4,],
      [4,8,7,8,1] ]
print("before transform:",X)
scaler=StandardScaler()
scaler.fit(X)
print("scale_ is :",scaler.scale_)
print("mean_ is :",scaler.mean_)
print("var_ is :",scaler.var_)
print("after transform:",scaler.transform(X))
```

结果如下：

```
before transform:[[1, 5, 1, 2, 10],[2, 6, 3, 2, 7],[3, 7, 5, 6, 4],[4, 8, 7, 8, 1]]
scale_ is : [ 1.11803399  1.11803399  2.23606798  2.59807621  3.35410197]
mean_ is : [ 2.5  6.5  4.   4.5  5.5]
var_ is : [  1.25   1.25   5.     6.75  11.25]
after transform:
[[-1.34164079 -1.34164079 -1.34164079 -0.96225045  1.34164079]
```

```
[-0.4472136  -0.4472136  -0.4472136  -0.96225045  0.4472136 ]
[ 0.4472136   0.4472136   0.4472136   0.57735027 -0.4472136 ]
[ 1.34164079  1.34164079  1.34164079  1.34715063 -1.34164079]]
```

其中

- ❑ scaler.scale_：存放的是每个属性的缩放倍数的倒数（其实就是每个属性的标准差）。
- ❑ scaler.mean_：存放的是每个特征的均值，这里依次为 2.5,6.5,4.,4.5,5.5。
- ❑ scaler.var_：存放的是每个特征的方差，这里依次为 1.25,1.25,5., 6.75,11.25。
- ❑ 在标准化后，每个特征的均值为 0，方差为 1。

11.3.4 正则化

scikit-learn提供的Normalizer能将数据正则化，其原型为：

```
class sklearn.preprocessing.Normalizer(norm='l2', copy=True)
```

参数

- ❑ norm：一个字符串，指定正则化方法。可以为如下。
  - ❍ l1：采用 L_1 范数正则化。
  - ❍ l2：采用 L_2 范数正则化。
  - ❍ max：采用 L_∞ 范数正则化。
- ❑ copy：一个布尔值，如果为True，则执行原地修改（节省空间，但是修改了原始数据）。

方法

- ❑ fit(X[, y])：不做任何事情，主要用于流水线Pipeline。
- ❑ transform(X[, y, copy])：将每一个样本正则化为范数等于单位 1。
- ❑ fit_transform(X[, y])：将每一个样本正则化为范数等于单位 1。

给出示例：

```
from sklearn.preprocessing import Normalizer
X=[   [1,2,3,4,5],
      [5,4,3,2,1],
      [1,3,5,2,4,],
      [2,4,1,3,5] ]
print("before transform:",X)
normalizer=Normalizer(norm='l2')
print("after transform:",normalizer.transform(X))
```

这里使用 L_2 范数。运行结果如下：

```
before transform:[[1, 1, 1, 1, 1], [2, 2, 2, 2, 2],[3, 3, 3, 3, 3],[4, 4, 4, 4, 4]]
```

```
after transform:
[[ 0.13483997  0.26967994  0.40451992  0.53935989  0.67419986]
 [ 0.67419986  0.53935989  0.40451992  0.26967994  0.13483997]
 [ 0.13483997  0.40451992  0.67419986  0.26967994  0.53935989]
 [ 0.26967994  0.53935989  0.13483997  0.40451992  0.67419986]]
```

可以看到在正则化后，每个样本的 L_2 范数为 1。

11.3.5 过滤式特征选取

VarianceThreshold

方差很小的属性，意味着该属性的识别能力很差。极端情况下，方差为 0，意味着该属性在所有样本上的值都是一个常数，可以通过scikit-learn提供的VarianceThreshold来剔除它。

VarianceThreshold的原型为：

```
class sklearn.feature_selection.VarianceThreshold(threshold=0.0)
```

参数

- threshold：一个浮点数，指定方差的阈值。低于此阈值的属性将被剔除。

属性

- variances_：一个数组，成员分别是各属性的方差。

方法

- fit(X[, y])：从样本数据中学习方差。
- transform(X)：执行特征选择（即删除低于指定阈值的属性）。
- fit_transform(X[, y])：从样本数据中学习方差，然后执行特征选择。
- get_support([indices])：如果indices=True，则返回被选出的特征的下标；如果indices =False，则返回一个布尔值组成的数组，该数组指示哪些特征被选择。
- inverse_transform(X)：根据被选出来的特征还原原始数据（特征选取的逆操作），但是对于被删除的属性值全部用 0 代替。

给出一个示例：

```
from sklearn.feature_selection import  VarianceThreshold
X=[[100,1,2,3],
   [100,4,5,6],
   [100,7,8,9],
   [101,11,12,13]]
selector=VarianceThreshold(1)
selector.fit(X)
```

```
print("Variances is %s"%selector.variances_)
print("After transform is %s"%selector.transform(X))
print("The surport is %s"%selector.get_support(True))
print("After reverse transform is %s"%
        selector.inverse_transform(selector.transform(X)))
```

特意将第一个属性的方差取得很小。结果如下：

```
Variances is [  0.1875  13.6875  13.6875  13.6875]
After transform is
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [11 12 13]]
The surport is [1 2 3]
After reverse transform is
[[ 0  1  2  3]
 [ 0  4  5  6]
 [ 0  7  8  9]
 [ 0 11 12 13]]
```

可以看到：

- selector.variances_依次给出四个属性的方差为0.1875,13.6875,13.6875,13.6875。
- 经过特征选取之后（selector.transform(X)），第一个属性被剔除了。
- 被选择出来的特征的下标为 1,2,3(selector.get_support(True))，表明第一个属性被提出来了。
- 根据特征选取之后的数据还原原始数据时发现，第一个属性（被剔除的属性）全部填充为零。

单变量特征提取

单变量特征提取通过计算每个特征的某个统计指标，然后根据该指标来选取特征。常用的类如下。

- SelectKBest：可以保留在该统计指标上得分最高的 k 个特征。
- SelectPercentile：可以保留在该统计指标上得分最高的百分之 k 的特征。

SelectKBest的原型为：

```
class sklearn.feature_selection.SelectKBest(score_func=<function f_classif>, k=10)
```

参数

- score_func：给出统计指标的函数，其参数为数组 X和数组y，返回值为(scores,pvalues)。sklearn提供的常用函数如下。

  - ❍ sklearn.feature_selection.f_regression：基于线性回归分析来计算统计指标。适用于回归问题。
  - ❍ sklearn.feature_selection.chi2：计算卡方统计量，适用于分类问题。
  - ❍ sklearn.feature_selection.f_classif：根据方差分析（Analysis of variance，ANOVA）的原理，依靠F-分布为概率分布的依据，利用平方和与自由度所计算的组间与组内均方估计出F值，适用于分类问题。
- ❑ k：一个整数或者字符串'all'，指定要保留最佳的几个特征。如果为'all'，则保留所有的特征。

属性

- ❑ scores_：一个数组，给出了所有特征的得分。
- ❑ pvalues_：一个数组，给出了所有特征得分的p-values。

方法

- ❑ fit(X,y)：从样本数据中学习统计指标得分。
- ❑ transform(X)：执行特征选择。
- ❑ fit_transform(X,y)：从样本数据中学习统计指标得分，然后执行特征选择。
- ❑ get_support([indices])：如果indices=True，则返回被选出的特征的下标；如果indices=False，则返回一个布尔值组成的数组，该数组指示哪些特征被选择。
- ❑ inverse_transform(X)：根据被选出来的特征还原原始数据（特征选取的逆操作），但是对于被删除的属性值全部用 0 代替。

SelectPercentile的原型为：

```
class sklearn.feature_selection.SelectPercentile(score_func=<function f_classif>,
          percentile=10)
```

参数

- ❑ score_func：给出统计指标的函数，其参数为数组 X和数组y，返回值为(scores,pvalues)。sklearn提供的常用函数如下。
  - ❍ sklearn.feature_selection.f_regression：基于线性回归分析来计算统计指标，适用于回归问题。
  - ❍ sklearn.feature_selection.chi2：计算卡方统计量，适用于分类问题。
  - ❍ sklearn.feature_selection.f_classif：根据方差分析Analysis of variance：ANOVA的原理，依靠F-分布为概率分布的依据，利用平方和与自由度所计算的组间与组内均方估计出F值，适用于分类问题。
- ❑ percentile：一个整数，指定要保留最佳的百分之几的特征，如10表示保留最佳的百分之十的特征。

属性

- ❑ scores_：一个数组，给出了所有特征的得分。
- ❑ pvalues_：一个数组，给出了所有特征得分的p-values。

方法

- ❑ fit(X,y)：从样本数据中学习统计指标得分。
- ❑ transform(X)：执行特征选择。
- ❑ fit_transform(X,y)：从样本数据中学习统计指标得分，然后执行特征选择。
- ❑ get_support([indices])：如果indices=True，则返回被选出的特征的下标；如果indices =False，则返回一个布尔值组成的数组，该数组指示哪些特征被选择。
- ❑ inverse_transform(X)：根据被选出来的特征还原原始数据（特征选取的逆操作），但对于被删除的属性值全部用 0 代替。

SelectKBest与SelectPercentile的用法类似。这里以SelectKBest为例：

```
from sklearn.feature_selection import SelectKBest,f_classif
X=[   [1,2,3,4,5],
      [5,4,3,2,1],
      [3,3,3,3,3,],
      [1,1,1,1,1] ]
y=[0,1,0,1]
print("before transform:",X)
selector=SelectKBest(score_func=f_classif,k=3)
selector.fit(X,y)
print("scores_:",selector.scores_)
print("pvalues_:",selector.pvalues_)
print("selected index:",selector.get_support(True))
print("after transform:",selector.transform(X))
```

这里一共有 5 个特征，选取f_classif指标最好的 3 个特征，运行结果为：

```
before transform: [[1, 2, 3, 4, 5], [5, 4, 3, 2, 1], [3, 3, 3, 3, 3], [1, 1, 1, 1, 1]]
scores_: [ 0.2  0.   1.   8.   9. ]
pvalues_: [ 0.69848865  1.          0.42264974  0.10557281  0.09546597]
selected index: [2 3 4]
after transform:
[[3 4 5]
 [3 2 1]
 [3 3 3]
 [1 1 1]]
```

可以看到：

- ❑ selector.scores_依次给出了 5 个属性的指标为0.2 0.   1.   8.   9.。

- 经过特征选取之后（selector.transform(X)），保留了最后三个特征。
- 被选择出来的特征下标为 2,3,4(selector.get_support(True))，表明最后三个特征被保留下来了。

11.3.6　包裹式特征选取

RFE

scikit-learn提供了RFE类来实现包裹式特征选取。RFE通过外部提供的一个学习器来选择特征。它要求学习器学习的是特征的权重（如线性模型），其原理如下。

- 首先：学习器在初始的特征集合以及初始的权重上训练。
- 然后：学习器学得每个特征的权重，剔除当前权重最小的那个特征，构成新的训练集。
- 再将学习器在新的训练集上训练，直到剩下的特征数量满足条件为止。

RFE的原型为：

```
class sklearn.feature_selection.RFE(estimator, n_features_to_select=None,
        step=1, estimator_params=None, verbose=0)
```

参数

- estimator：一个学习器，它必须提供一个.fit方法和一个.coef_属性，其中.coef_属性中存放的是学习到的各特征的权重系数。通常使用SVM和广义线性模型作为estimator参数。
- n_features_to_select：一个整数或者None，指定要选出几个特征。如果为None，则默认选取一半的特征。
- step：一个整数或者浮点数，指定每次迭代要剔除权重最小的几个特征。
  - 如果大于等于 1，则作为整数，指定每次迭代要剔除权重最小的特征的数量。
  - 如果在0.0~1.0之间，则指定每次迭代要剔除权重最小的特征的比例。
- estimator_params：一个字典，用于设定estimator的参数。该参数将被移除，推荐使用estimator.set_params()方法直接设定estimator的参数。
- verbose：一个整数，控制输出日志。

属性

- n_features_：一个整数，给出了被选出的特征的数量。
- support_：一个数组，给出了被选择特征的mask。
- ranking_：特征排名，被选出特征的排名为 1。

方法

- ❑ fit(X,y)：训练RFE模型。
- ❑ transform(X)：执行特征选择。
- ❑ fit_transform(X,y)：从样本数据中学习RFE模型，然后执行特征选择。
- ❑ get_support([indices])：如果indices=True，则返回被选出的特征的下标；如果indices =False，则返回一个布尔值组成的数组，该数组指示哪些特征被选择。
- ❑ inverse_transform(X)：根据被选出来的特征还原原始数据（特征选取的逆操作），但对于被删除的属性值全部用 0 代替。
- ❑ predict(X)/predict_log_proba(X) /predict_proba(X)：将X进行特征选择之后，再使用内部的estimator来预测。
- ❑ score(X, y)：将X进行特征选择之后，再使用内部的estimator来评分。

给出一个示例：

```
from sklearn.feature_selection import  RFE
from sklearn.svm import LinearSVC
from sklearn.datasets import  load_iris
iris=load_iris()
X=iris.data
y=iris.target
estimator=LinearSVC()
selector=RFE(estimator=estimator,n_features_to_select=2)
selector.fit(X,y)
print("N_features %s"%selector.n_features_)
print("Support is %s"%selector.support_)
print("Ranking %s"%selector.ranking_)
```

运行结果为：

```
N_features 2
Support is [False  True False  True]
Ranking [3 1 2 1]
```

可以看到：

- ❑ selector.n_features_给出了最终挑选出来的特征数量为 2。
- ❑ selector.support_给出了每个特征被挑选出来与否。如第二个与第四个特征对应的mask为True，则挑选出了第二个特征与第四个特征。
- ❑ selector.ranking_给出了最终各特征的排名。如第二个与第四个特征对应的rank为1，则挑选出了第二个特征与第四个特征。

注意：特征提取对于预测性能的提升没有必然的联系。这里可以比较：

```
from sklearn.feature_selection import  RFE
from sklearn.svm import LinearSVC
from  sklearn import  cross_validation
from sklearn.datasets import  load_iris

### 加载数据
iris=load_iris()
X,y=iris.data,iris.target
### 特征提取
estimator=LinearSVC()
selector=RFE(estimator=estimator,n_features_to_select=2)
X_t=selector.fit_transform(X,y)
#### 切分测试集与验证集
X_train,X_test,y_train,y_test=cross_validation.train_test_split(X, y,
            test_size=0.25,random_state=0,stratify=y)
X_train_t,X_test_t,y_train_t,y_test_t=cross_validation.train_test_split(X_t, y,
            test_size=0.25,random_state=0,stratify=y)
### 测试与验证
clf=LinearSVC()
clf_t=LinearSVC()
clf.fit(X_train,y_train)
clf_t.fit(X_train_t,y_train_t)
print("Original DataSet: test score=%s"%(clf.score(X_test,y_test)))
print("Selected DataSet: test score=%s"%(clf_t.score(X_test_t,y_test_t)))
```

运行结果如下：

```
Original DataSet: test score=0.974358974359
Selected DataSet: test score=0.948717948718
```

可以看到：原始数据利用支持向量机预测的准确率高达 97.436%，而经过特征提取之后的数据利用支持向量机预测的准确率为 94.872 %。之所以特征提取之后的预测准确率降低，是因为被剔除的特征中包含了有效的信息。因此抛弃了这部分信息会在一定程度上降低预测准确率。

RFECV

scikit-learn还提供了RFECV类，它是RFE的一个变体，它执行一个交叉验证来寻找最优的剩余特征数量，因此不需要指定保留多少个特征。其原型为：

```
class sklearn.feature_selection.RFECV(estimator, step=1, cv=None, scoring=None,
 estimator_params=None, verbose=0)
```

参数

- ❑ estimator:一个学习器,它必须提供一个.fit方法和一个.coef_属性,其中.coef_属性中存放的是学习到的各特征的权重系数。通常使用SVM和广义线性模型作为estimator参数。
- ❑ step：一个整数或者浮点数，指定每次迭代要剔除权重最小的几个特征。
  - ❍ 如果大于等于 1，则作为整数，指定每次迭代要剔除权重最小的特征的数量。
  - ❍ 如果在0.0~1.0之间，则指定每次迭代要剔除权重最小的特征的比例。
- ❑ cv：一个整数，或者交叉验证生成器或者一个可迭代对象，它决定了交叉验证策略。
  - ❍ 如果为None,则使用默认的3折交叉验证。
  - ❍ 如果为整数 k ，则使用 k 折交叉验证。
  - ❍ 如果为交叉验证生成器，则直接使用该对象。
  - ❍ 如果为可迭代对象，则使用该可迭代对象迭代生成训练-测试集合。
- ❑ estimator_params：一个字典，用于设定estimator的参数。该参数将被移除，推荐使用estimator.set_params()方法直接设定estimator的参数。
- ❑ scoring：一个字符串或者可调用对象，用于评估底层estimator的预测性能。如果为None，则使用estimator.score()方法。
- ❑ verbose：一个整数，控制输出日志。

属性

- ❑ n_features_：一个整数，给出了被选出特征的数量。
- ❑ support_：一个数组，给出了被选择特征的mask。
- ❑ ranking_：特征排名，被选出特征的排名为 1。
- ❑ grid_scores_：一个数组，给出了交叉验证的预测性能得分，其成员为每个特征子集执行交叉验证后的预测得分。

方法

- ❑ fit(X,y)：训练RFECV模型。
- ❑ transform(X)：执行特征选择。
- ❑ fit_transform(X,y)：从样本数据中学习RFECV模型，然后执行特征选择。
- ❑ get_support([indices])：如果indices=True，则返回被选出的特征的下标；如果indices=False，则返回一个布尔值组成的数组，该数组指示哪些特征被选择。
- ❑ inverse_transform(X)：根据被选出来的特征还原原始数据（特征选取的逆操作），但对于被删除的属性值全部用 0 代替。
- ❑ predict(X)/predict_log_proba(X) /predict_proba(X)：将X进行特征选择之后，再使用内部的estimator来预测。
- ❑ score(X, y)：将X进行特征选择之后，再使用内部的estimator来评分。

给出一个示例：

```
import numpy as np
from sklearn.feature_selection import  RFECV
from sklearn.svm import LinearSVC
from sklearn.datasets import  load_iris
iris=load_iris()
X=iris.data
y=iris.target
estimator=LinearSVC()
selector=RFECV(estimator=estimator,cv=3)
selector.fit(X,y)
print("N_features %s"%selector.n_features_)
print("Support is %s"%selector.support_)
print("Ranking %s"%selector.ranking_)
print("Grid Scores %s"%selector.grid_scores_)
```

运行结果为：

```
N_features 4
Support is [ True  True  True  True]
Ranking [1 1 1 1]
Grid Scores [ 0.91421569  0.94689542  0.95383987  0.96691176]
```

可以看到：

- selector.n_features_给出了最终挑选出来的特征数量为 4（即所有的特征都保留了）。
- selector.support_给出了每个特征被挑选出来与否。这里所有特征对应的mask都是 True，因此所有的特征都被选中。
- selector.ranking_给出了最终各特征的排名。这里所有的特征对应的rank为1，因此所有的特征都被选中。
- selector.grid_scores依次给出了单个特征上交叉验证得到的最佳预测准确率。这里四个特征依次对应的交叉验证最佳预测准确率为 0.91421569 0.94689542 0.95383987 0.96691176。

11.3.7 嵌入式特征选取

scikit-learn提供了SelectFromModel来实现嵌入式特征选取。SelectFromModel使用外部提供的estimator来工作。estimator必须有coef_或者feature_importances_属性。当某个特征对应的coef_或者feature_importances_低于某个阈值时，该特征将被移除。当然你可以不指定阈值，而使用启发式的方法，如指定均值mean，指定中位数median或者指定这些统计量的一个倍数，如0.1*mean。

SelectFromModel的原型为：

```
class sklearn.feature_selection.SelectFromModel(estimator, threshold=None,
        prefit=False)
```

参数

- estimator:一个学习器,它可以是未训练的(prefit=False),或者是已经训练好的(prefit=True)。
- threshold：一个字符串或者浮点数或者None，指定特征重要性的一个阈值。低于此阈值的特征将被剔除。
  - 如果为浮点数，则指定阈值的绝对大小。
  - 如果为字符串，可以为如下。
    - 'mean'：阈值为特征重要性的均值。
    - 'median'：阈值为特征重要性的中值。
    - 如果是'1.5*mean'，则表示阈值为 1.5 倍的特征重要性的均值。
  - 如果为None：
    - 如果estimator有一个penalty参数，且该参数设置为'l1'，则阈值默认为1e-5；
    - 其他情况下，阈值默认为'mean'。
- prefit:一个布尔值，指定estimator是否已经训练好了。如果prefit=False，则estimator是未训练的。

属性

- threshold_：一个浮点数，存储了用于特征选取重要性的阈值。

方法

- fit(X,y)：训练SelectFromModel模型。
- transform(X)：执行特征选择。
- fit_transform(X,y)：从样本数据中学习RFE模型，然后执行特征选择。
- get_support([indices])：如果indices=True，则返回被选出的特征的下标；如果indices=False，则返回一个布尔值组成的数组，该数组指示哪些特征被选择。
- inverse_transform(X)：根据被选出来的特征还原原始数据（特征选取的逆操作），但对于被删除的属性值全部用 0 代替。
- partial_fit(X[, y])：只训练SelectFromModel模型一次。

给出示例：

```
import numpy as np
from sklearn.feature_selection import  SelectFromModel
from sklearn.svm import LinearSVC
from sklearn.datasets import  load_digits
digits=load_digits()
```

```
X=digits.data
y=digits.target
estimator=LinearSVC(penalty='l1',dual=False)
selector=SelectFromModel(estimator=estimator,threshold='mean')
selector.fit(X,y)
selector.transform(X)
print("Threshold %s"%selector.threshold_)
print("Support is %s"%selector.get_support(indices=True))
```

在这里指定阈值为'mean'，表示特征重要性的均值。运行结果如下：

```
Threshold 0.683248110816
Support is [ 2  3  4  5  6  9 12 14 16 18 19 20 21 22 24 26 27 30 33
    36 38 41 42 43 44 45 53 54 55 61]
```

可以看到：

- selector.threshold_给出了最终用于特征提取的特征重要性的阈值，这里为 0.683248110816。
- .get_support方法给出了被选中的特征的下标，这里特征的总数量为 64，所以下标从0~63。

这里重点说明estimator的类型。

- 线性且带有L1正则化项的模型。此时estimator学习的模型具有较好的系数解：大量的系数为零。
    - 对于回归问题，可以使用linear_model.Lasso类型的estimator。
        - 在Lasso中，α 参数控制了稀疏性：如果 α 越小，则稀疏性越小，则被选择的特征越多；如果 α 越大，则稀疏性越大，则被选择的特征越少。
    - 对于分类问题，可以使用linear_model.LogisticRegression或者svm.LinearSVC类型的estimator。
        - 在SVM和logistic-regression中，参数C控制了稀疏性：如果C越小，稀疏性越大，则被选择的特征越少；如果C越大，稀疏性越小，则被选择的特征越多。

> 参数的选取应该是以最终预测性能最佳为目的，而不是为了选取最少的特征为目的。

- 基于决策树的模型（在sklearn.tree模块中）和基于森林的模型（在sklearn.ensemble模块中）。这些模型能计算特征的重要性并用于特征选取。

给出下面代码来说明 α 和 C 参数与稀疏性的关系。通过权重系数中等于零的系数的个数来表征稀疏性：零的个数越多，稀疏性越大！

```
import numpy as np
import  matplotlib.pyplot as plt
from sklearn.svm import LinearSVC
from sklearn.datasets import  load_digits,load_diabetes
from sklearn.linear_model import Lasso
def test_Lasso(*data):
    X,y=data
    alphas=np.logspace(-2,2)
    zeros=[]
    for alpha in alphas:
        regr=Lasso(alpha=alpha)
        regr.fit(X,y)
        ### 计算零的个数 ###
        num=0
        for ele in regr.coef_:
            if abs(ele) < 1e-5:num+=1
        zeros.append(num)
    ##### 绘图
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    ax.plot(alphas,zeros)
    ax.set_xlabel(r"$\alpha$")
    ax.set_xscale("log")
    ax.set_ylim(0,X.shape[1]+1)
    ax.set_ylabel("zeros in coef")
    ax.set_title("Sparsity In Lasso")
    plt.show()
def test_LinearSVC(*data):
    X,y=data
    Cs=np.logspace(-2,2)
    zeros=[]
    for C in Cs:
        clf=LinearSVC(C=C,penalty='l1',dual=False)
        clf.fit(X,y)
         ### 计算零的个数 ###
        num=0
        for row in clf.coef_:
            for ele in row:
                if abs(ele) < 1e-5:num+=1
        zeros.append(num)
    ##### 绘图
    fig=plt.figure()
    ax=fig.add_subplot(1,1,1)
    ax.plot(Cs,zeros)
    ax.set_xlabel("C")
```

```
    ax.set_xscale("log")
    ax.set_ylabel("zeros in coef")
    ax.set_title("Sparsity In SVM")
    plt.show()
if __name__=='__main__':
    data=load_diabetes()
    test_Lasso(data.data,data.target)
    data=load_digits()
    test_LinearSVC(data.data,data.target)
```

可以看到，正如前文所述：

- 在Lasso中，α 参数控制了稀疏性：如果 α 越小，则稀疏性越小；如果 α 越大，则稀疏性越大，如图 11.3 所示。

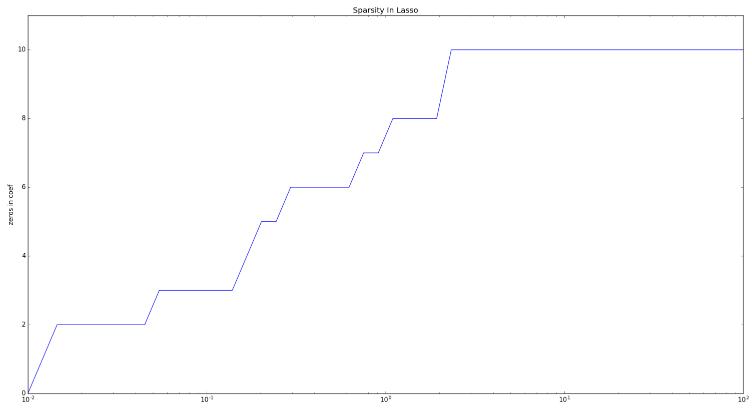


图 11.3　Sparsity_In_Lasso

- 在SVM和logistic-regression中，参数C控制了稀疏性：如果C越小，则稀疏性越大；如果C越大，则稀疏性越小，如图 11.4 所示。

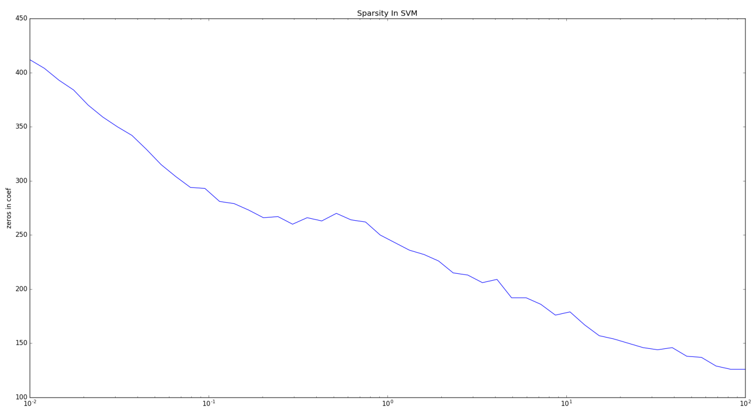


图 11.4　Sparsity_In_SVM

11.3.8 学习器流水线（Pipeline）

scikit-learn提供了Pipeline来将多个学习器组成流水线。通常流水线的形式为：将数据标准化的学习器 --> 特征提取的学习器 --> 执行预测的学习器。除了最后一个学习器之外，前面的所有学习器必须提供transform方法，该方法用于数据变换（如归一化、正则化，以及特征提取等）。

Pipeline的原型为：

```
class sklearn.pipeline.Pipeline(steps)
```

参数

- steps：一个列表，列表的元素为(name,transform)元组，其中name是学习器的名字，用于输出和日志；transform是学习器，之所以叫transform是因为这个学习器（除了最后一个学习器）必须提供transform方法。

属性

- named_steps:一个字典,字典的键就是steps中各元组的name元素,字典的值就是steps中各元组的transform元素。

方法

- fit(X[, y])：启动流水线，依次对各个学习器（除了最后一个学习器）执行.fit方法和.transform方法转换数据，最后一个学习器执行.fit方法训练学习器。
- transform(X)：启动流水线，依次对各个学习器执行.fit方法和.transform方法转换数据。要求每个学习器都实现了.transform方法。
- fit_transform(X[, y])：启动流水线，依次对各个学习器（除了最后一个学习器）执行.fit方法和.transform方法转换数据，最后一个学习器执行.fit_transform方法转换数据。
- inverse_transform(X)：将转换后的数据逆转换成原始数据，要求每个学习器都实现了.inverse_transform方法。
- predict(X)/predict_log_proba(X) /predict_proba(X)：将X进行数据转换后，用最后一个学习器来预测。
- score(X, y)：将X进行数据转换后，用最后一个学习器来给出预测评分。

给出一个示例：

```
from sklearn.svm import  LinearSVC
from sklearn.datasets import  load_digits
from sklearn import  cross_validation
from sklearn.linear_model import LogisticRegression
from  sklearn.pipeline import Pipeline
def test_Pipeline(data):
```

```
    X_train,X_test,y_train,y_test=data
    steps=[("Linear_SVM",LinearSVC(C=1,penalty='l1',dual=False)),
           ("LogisticRegression",LogisticRegression(C=1))]
    pipeline=Pipeline(steps)
    pipeline.fit(X_train,y_train)
    print("Named steps:",pipeline.named_steps)
    print("Pipeline Score:",pipeline.score(X_test,y_test))
if __name__=='__main__':
    data=load_digits()
    X=data.data
    y=data.target
    test_Pipeline(cross_validation.train_test_split(X, y,test_size=0.25
            ,random_state=0,stratify=y))
```

运行结果如下：

```
Named steps: {'LogisticRegression': LogisticRegression(C=1, class_weight=None,
        dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False), 'Linear_SVM': LinearSVC(C=1,
    class_weight=None, dual=False, fit_intercept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l1', random_state=None, tol=0.0001,
     verbose=0)}
Pipeline Score: 0.949002217295
```

可以看到：

- pipeline.named_steps给出了流水线上每一步使用的学习器；
- pipeline.score给出了最后一个学习器的预测性能得分（这里为 0.949002217295），该学习器的输入数据为原始数据经过流水线处理后的数据。

11.3.9 字典学习

DictionaryLearning

scikit-learn提供了DictionaryLearning用于字典学习，其原型为：

```
class sklearn.decomposition.DictionaryLearning(n_components=None, alpha=1,
 max_iter=1000, tol=1e-08, fit_algorithm='lars', transform_algorithm='omp',
 transform_n_nonzero_coefs=None, transform_alpha=None, n_jobs=1,
 code_init=None, dict_init=None, verbose=False, split_sign=False, random_state=None)
```

参数

- n_components: 一个整数，指定了字典大小 k。
- alpha：一个浮点数，指定了 L_1 正则化项的系数 λ，它控制了稀疏性。
- max_iter：一个整数，指定了最大迭代次数。
- tol：一个浮点数，指定了收敛阈值。
- fit_algorithm：一个字符串，指定了求解算法，可以为下列值。
  - 'lars'：使用least angle regression算法来求解。
  - 'cd'：使用coordinate descent算法来求解。
- transform_algorithm：一个字符串，指定了数据转换的方法，可以为下列值。
  - 'lasso_lars'：使用Lars算法来求解。
  - 'lasso_cd'：使用coordinate descent算法来求解。
  - 'lars'：使用least angle regression算法来求解。
  - 'omp'：使用正交匹配的方法来求解。
  - 'threshold'：通过字典转换后的坐标中，小于 transform_alpha 的属性值都设成 0。
- transform_n_nonzero_coefs：一个整数，指定了解中每一列中非零元素的个数。只用于lars算法和omp算法（omp算法中，可能被transform_alpha参数覆盖），默认为 0.1*n_features。
- transform_alpha：一个浮点数，默认为 1.0。
  - 如果算法为lasso_lars或者lasso_cd，则该参数指定了L1正则化项的系数。
  - 如果算法为threshold，则该参数指定了属性为 0 的阈值。
  - 如果算法为omp，则该参数指定了重构误差的阈值，此时它覆盖了 transform_n_nonzero_coefs 参数。
- split_sign：一个布尔值，指定是否拆分系数特征向量为其正向值和负向值的拼接。
- n_jobs：一个整数，指定并行性。如果为 −1，则表示将训练和预测任务派发到所有 CPU 上。
- code_init：一个数组，指定了初始编码，它用于字典学习算法的热启动。
- dict_init：一个数组，指定了初始字典，它用于字典学习算法的热启动。
- verbose：一个整数，控制输出日志。
- random_state: 一个整数，或者一个RandomState 实例，或者None。
  - 如果为整数，则它指定了随机数生成器的种子。
  - 如果为RandomState实例，则指定了随机数生成器。
  - 如果为None，则使用默认的随机数生成器。

属性

- components_：一个数组，存放学到的字典。
- error_：一个数组，存放每一轮迭代的误差。

- n_iter_：一个整数，存放迭代的次数。

方法

- fit(X,y)：学习字典。
- transform(X)：根据学到的字典进行编码。
- fit_transform(X,y)：学习字典并执行字典编码。

给出一个示例：

```
from sklearn.decomposition import DictionaryLearning
X=[[1,2,3,4,5],
   [6,7,8,9,10],
   [10,9,8,7,6,],
   [5,4,3,2,1] ]
print("before transform:",X)
dct=DictionaryLearning(n_components=3)
dct.fit(X)
print("components is :",dct.components_)
print("after transform:",dct.transform(X))
```

运行结果为：

```
before transform:[[1, 2, 3, 4, 5],[6, 7, 8, 9, 10],[10, 9, 8, 7, 6],[5, 4, 3, 2, 1]]
components is :
 [[-0.54734989 -0.49408715 -0.44082441 -0.38756167 -0.33429893]
 [-0.67419986 -0.53935989 -0.40451992 -0.26967994 -0.13483997]
 [ 0.18512637  0.30070081  0.41627526  0.53184971  0.64742416]]
after transform:
[[  0.           0.           7.39987341]
 [  0.           0.          17.80675497]
 [-18.16560383   0.           0.        ]
 [  0.          -7.41619849   0.        ]]
```

可以看到：

- 原始数据每个样本的特征数量为 5，经过字典学习后，转换后每个样本的特征数量为 3。这是因为我们指定了字典大小 $k=3$。
- dct.components_存放了模型学习到的字典。由于转换前样本特征数量为 5，转换后每个样本的特征数量为 3，因此字典为 5×3 大小的矩阵。

MiniBatchDictionaryLearning

MiniBatchDictionaryLearning也是字典学习，它主要用于大规模数据。它每次训练一批样本，然后连续多次训练。它的接口类似于DictionaryLearning，其原型为：

```
class sklearn.decomposition.MiniBatchDictionaryLearning(n_components=None, alpha=1,
 n_iter=1000, fit_algorithm='lars', n_jobs=1, batch_size=3, shuffle=True,
 dict_init=None, transform_algorithm='omp', transform_n_nonzero_coefs=None,
 transform_alpha=None, verbose=False, split_sign=False, random_state=None)
```

参数

- n_components：一个整数，指定了字典大小 k。
- alpha：一个浮点数，指定了 L_1 正则化项的系数 λ，它控制了稀疏性。
- n_iter：一个整数，指定了总的执行迭代的数量。
- fit_algorithm：一个字符串，指定了求解算法，可以为下列值。
    - 'lars'：使用least angle regression算法来求解。
    - 'cd'：使用coordinate descent算法来求解。
- transform_algorithm：一个字符串，指定了数据转换的方法，可以为下列值。
    - 'lasso_lars'：使用Lars算法来求解。
    - 'lasso_cd'：使用coordinate descent算法来求解。
    - 'lars'：使用least angle regression算法来求解。
    - 'omp'：使用正交匹配的方法来求解。
    - 'threshold'：通过字典转换后的坐标中，将小于 transform_alpha 的属性值都设成 0。
- transform_n_nonzero_coefs：一个整数，指定了解中每一列中非零元素的个数，只用于lars算法和omp算法（omp算法中，可能被transform_alpha参数覆盖），默认为 0.1*n_features。
- transform_alpha：一个浮点数，默认为 1.0。
    - 如果算法为lasso_lars或者lasso_cd，则该参数指定了L1正则化项的系数。
    - 如果算法为threshold，则该参数指定了属性为 0 的阈值。
    - 如果算法为omp，则该参数指定了重构误差的阈值，此时它覆盖了 transform_n_nonzero_coefs 参数。
- split_sign：一个布尔值，指定是否拆分系数特征向量为其正向值和负向值的拼接。
- n_jobs：一个整数，指定并行性。如果为 −1，则表示将训练和预测任务派发到所有 CPU 上。
- dict_init：一个数组，指定了初始字典，它用于字典学习算法的热启动。
- verbose：一个整数，控制输出日志。
- batch_size：一个整数，指定了每次训练时的样本数量。
- shuffle：一个布尔值，指定在训练每一批样本之前，是否对该批次样本进行混洗。
- random_state：一个整数，或者一个RandomState 实例，或者None。
    - 如果为整数，则它指定了随机数生成器的种子。
    - 如果为RandomState实例，则指定了随机数生成器。

- 如果为None，则使用默认的随机数生成器。

属性

- components_：一个数组，存放学到的字典。
- inner_stats_：数组的元组，存放算法的中间状态。
- n_iter_：一个整数，存放迭代的次数。

方法

- fit(X,y)：学习字典。
- transform(X)：根据学到的字典进行编码。
- fit_transform(X,y)：学习字典并执行字典编码。
- partial_fit(X[, y, iter_offset])：只训练一个批次的样本。

第 12 章

模型评估、选择与验证

12.1 概述

机器学习已经成为我们生活中的一部分，对购买者、消费者或是希望进行研究和实践的人员都很重要！无论我们应用预测建模技术来进行研究还是解决业务问题，都有一个共同点：那就是要做“好”的预测！机器学习的目的是使学习到的模型不仅对已知数据而且对未知数据都有很好的预测能力。

诚然，需要训练数据模型，但是我们怎么知道它能很好地在看不见的数据中运行？怎么避免它仅仅记忆那些我们提供的数据，却未能对我们从未见过的样本作出良好的预测？我们如何合理评估、优先选择和有效验证一个好的模型呢？总之，评价一个机器学习模型的好坏需要特定的评估方法，并据此对模型进行选择，从而得到一个更好的模型。

通常机器学习过程包括两个阶段，原型设计阶段和应用阶段。

原型设计阶段是使用历史数据训练一个适合解决目标任务的一个或多个机器学习模型，并对模型进行验证（Validation）与离线评估（Offline evalutation），然后通过评估指标选择一个较好的模型。比如在分类任务中，选择一个适合自己问题的最好的分类算法。

应用阶段是当模型达到设定的指标值时便将模型上线，投入生产，使用新生成的数据来对该模型进行在线评估（Online evalution），以及使用新数据更新模型。在对模型进行离线评估或者在线评估时，它们所用的评价指标往往不同。比如在离线评估中，我们经常使用准确率（accuracy）、精确率－召回率（precision － recall），而在在线评估中，一般使用一些商业评价指标，如用户生命周期值（customer lifetime value）、广告点击率（click through rate）、用户流失率（customer churn rate）等，这些指标才是模型使用者最终关心的一些指标，甚至在对模型进行训练和验证过程中使用的评价指标都不一样。

12.2 算法笔记精华

12.2.1 损失函数和风险函数

损失函数

对于给定的输入样本 $\vec{x}$，设其真实值为 y。由决策函数 $\hat{y}=f(\vec{x})$ 或者 $\hat{y}=\arg\max_y P(y/\vec{x})$ 预测的输出值 $\hat{y}$ 与真实值 y 可能不一致。我们用损失函数度量错误的程度，也就是损失的程度，记作 $L(y,f(\vec{x}))$ 或者 $L(y,P(y/\vec{x}))$。常用的损失函数如下。

- 0-1 损失函数：

$$L(y,f(\vec{x}))=\begin{cases}1, & \text{if } y\neq f(\vec{x})\\ 0, & \text{if } y=f(\vec{x})\end{cases}$$

- 平方损失函数：$L(y,f(\vec{x}))=(Y-f(\vec{x}))^2$
- 绝对损失函数：$L(y,f(\vec{x}))=|y-f(\vec{x})|$
- 对数损失函数：$L(y,P(y/\vec{x}))=-\log P(y/\vec{x})$

> 因为 $(\vec{x},y)$ 已经出现，因此理论上 $P(y/\vec{x})$ 为 1。如果它不为 1，则说明预测有误差。越远离 1，说明误差越大。

上面给出单个样本的损失函数。对于样本集合，其损失函数等于所有样本的损失之和。

风险函数和经验风险

通常损失函数值越小，模型就越好。假设 $(\vec{x},y)$ 服从联合分布 $P(\vec{x},y)$，定义风险函数为损失函数的期望：

$$R_{exp}(f)=E_P\left[L(y,f(\vec{x}))\right]=\int_{\mathcal{X}\times\mathcal{Y}}L(y,f(\vec{x}))P(\vec{x},y)\mathrm{d}\vec{x}\mathrm{d}y$$

其中 $\mathcal{X},\mathcal{Y}$ 分别为输入空间和输出空间。学习的目标是选择风险函数最小的模型。但是问题是 $P(\vec{x},y)$ 未知。实际上如果它已知，则可以轻而易举地求得条件概率分布，也就不需要学习了。

给定训练集 $T=\{(\vec{x}_1,y_1),(\vec{x}_2,y_2),\cdots,(\vec{x}_N,y_N)\}$，模型 $f(X)$ 关于 T 的经验风险定义为：

$$R_{emp}(f)=\frac{1}{N}\sum_{i=1}^{N}L(y_i,f(\vec{x}_i))$$

经验风险是模型在 T 上的平均损失。根据大数定律，当 $N\to\infty$ 时，$R_{emp}(f)\to R_{exp}(f)$。

但是由于现实中训练集中样本数量有限，所以需要对经验风险进行矫正。有以下两种常用的策略。

- 经验风险最小化(Empirical Risk Minimization，ERM)：经验风险最小的模型就是最优的模型。即

$$\min_{f\in\mathcal{F}}\frac{1}{N}\sum_{i=1}^{N}L(y_i,f(\vec{\boldsymbol{x}}_i))$$

- 结构风险最小化(Structurel Risk Minimization，SRM)：它是在经验风险上加上表示模型复杂度的正则化项（或者称之为罚项）。这是为了防止过拟合而提出的。结构风险定义为

$$R_{srm}(f)=\frac{1}{N}\sum_{i=1}^{N}L(y_i,f(\vec{\boldsymbol{x}}_i))+\lambda J(f)$$

  其中 $J(f)$ 为模型复杂度，是定义在假设空间 $\mathcal{F}$ 上的泛函。f 越复杂，则 $J(f)$ 越大。$\lambda\geqslant 0$ 为系数，用于对经验风险和模型复杂度进行折中。SRM策略认为，结构风险最小的模型是最优的模型：

$$\min_{f\in\mathcal{F}}\left[\frac{1}{N}\sum_{i=1}^{N}L(y_i,f(\vec{\boldsymbol{x}}_i))+\lambda J(f)\right]$$

极大似然估计就是经验风险最小化的例子。极大似然估计：已知样本集 $T=\{(\vec{\boldsymbol{x}}_1,y_1),(\vec{\boldsymbol{x}}_2,y_2),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\}$，则出现这种训练集的概率为 $\prod_{i=1}^{N}P(y_i/\vec{\boldsymbol{x}}_i)$，则根据 T 出现概率最大为：

$$\max\prod_{i=1}^{N}P(y_i/\vec{\boldsymbol{x}}_i)\to\max\sum_{i=1}^{N}\log P(y_i/\vec{\boldsymbol{x}}_i)$$

$$\to\min\sum_{i=1}^{N}(-\log P(y_i/\vec{\boldsymbol{x}}_i))$$

$$def\ :\ L(Y,P(Y/X))=-\log P(Y/X)$$

$$\min\sum_{i=1}^{N}(-\log P(y_i/\vec{\boldsymbol{x}}_i))\to\min\sum_{i=1}^{N}L(y_i,P(y_i/\vec{\boldsymbol{x}}_i))$$

$$\to\min\frac{1}{N}\sum_{i=1}^{N}L(y_i,P(y_i/\vec{\boldsymbol{x}}_i))$$

即经验风险最小化 ERM。

最大后验概率估计就是结构风险最小化的例子。已知样本集 $T=\{(\vec{\boldsymbol{x}}_1,y_1),(\vec{\boldsymbol{x}}_2,y_2),\cdots,(\vec{\boldsymbol{x}}_N,y_N)\}$，假设已知参数 $\boldsymbol{\theta}$ 的先验分布为 $g(\theta)$，出现这种训练集的概率为 $\prod_{i=1}^{N}P(y_i/\vec{\boldsymbol{x}}_i)g(\theta)$，则根据 T 出现概率最大为：

$$\max\prod_{i=1}^{N}P(y_i/\vec{\boldsymbol{x}}_i)g(\theta)\rightarrow\max\sum_{i=1}^{N}\log P(y_i/\vec{\boldsymbol{x}}_i)+\log g(\theta)$$

$$\rightarrow\min\sum_{i=1}^{N}(-\log P(y_i/\vec{\boldsymbol{x}}_i))+\log\frac{1}{g(\theta)}$$

$$def\ :\ L(Y,P(Y/X))=-\log P(Y/X),\lambda=\frac{1}{N},J(f)=\log\frac{1}{g(\theta)}$$

$$\min\sum_{i=1}^{N}(-\log P(y_i/\vec{\boldsymbol{x}}_i))+\log\frac{1}{g(\theta)}\rightarrow\min\sum_{i=1}^{N}L(y_i,P(y_i/\vec{\boldsymbol{x}}_i))+J(f)$$

$$\rightarrow\min\frac{1}{N}\sum_{i=1}^{N}L(y_i,P(y_i/\vec{\boldsymbol{x}}_i))+\lambda J(f)$$

即结构风险最小化 SRM。

12.2.2 模型评估方法

训练误差与测试误差

训练误差：关于训练集的平均损失。假设学习到的模型为 $y=\hat{f}(\vec{\boldsymbol{x}})$，则训练误差为：

$$R_{emp}(\hat{f})=\frac{1}{N}\sum_{i=1}^{N}L(y_i,\hat{f}(\vec{\boldsymbol{x}}_i))$$

其中 N 为训练样本容量。训练误差的大小虽然有意义，但是本质上不重要。

测试误差：定义模型关于测试集的平均损失为：

$$e_{test}=\frac{1}{N'}\sum_{i=1}^{N'}L(y_i,\hat{f}(\vec{\boldsymbol{x}}_i))$$

其中 N' 为测试样本容量。测试误差反映了学习方法对未知测试数据集的预测能力。

泛化误差

模型的泛化能力指的是学到的模型对未知数据的预测能力。泛化误差越小，该模型越有效。泛化误差定义为所学习模型的期望风险，即：

$$R_{exp}(\hat{f})=E[L(y,\hat{f}(\vec{\boldsymbol{x}}))]=\int_{\mathcal{X}\times\mathcal{Y}}L(y,\hat{f}(\vec{\boldsymbol{x}}))P(\vec{\boldsymbol{x}},y)\mathrm{d}\vec{\boldsymbol{x}}\ \mathrm{d}y$$

通常泛化误差是不可知的，因为无法获取联合概率分布 $P(\vec{x}, y)$ 以及无限的采样点。

泛化误差的概率上界简称为泛化误差上界。通常借助于比较两个模型的泛化误差上界来比较它们的优劣。

过拟合

过拟合（over-fitting）指的是学习时选择的模型包含的参数过多，以至于出现该模型对于已知数据预测得很好，而对于未知数据预测得很差的现象。过拟合的原因是将训练样本本身的一些特点当作了所有潜在样本都具有的一般性质，这会造成泛化能力下降。过拟合无法避免，只能缓解。与过拟合对应的是“欠拟合”（underfitting）。欠拟合的原因一般是学习能力低下造成的。

常用的防止过拟合的办法为正则化。正则化是基于结构化风险最小化（SRM）策略的实现。在不同的问题中，正则化项可以有以下两种不同的形式。

- 回归问题中：损失函数是平方损失，正则化项是参数向量的 L_2 范数。
- 贝叶斯估计中：正则化项对应于模型的先验概率 $\log \frac{1}{g(\theta)}$。

12.2.3 模型评估

通常用测试集对学习方法进行评估，这是因为我们需要挑选泛化能力最强的那个模型。评估的方法有留出法、交叉验证法、留一法、自助法等。

留出法

留出法 (hold-out)：直接将数据切分为三个互斥的部分（也可以切分成两部分，此时训练集也是验证集），然后在训练集上训练模型，在验证集上选择模型，最后用测试集上的误差作为泛化误差的估计。

数据集的划分要尽可能保持数据分布的一致性，如在分类任务中至少要保持样本的类别比例相似。此时可以使用分层采样来保留各个集合中的类别比例。若训练集、验证集、测试集中类别比例差别很大，将由于训练/验证/测试数据分布的差异而导致误差估计偏差。

即使进行了分层采样，仍然存在多种划分方式，从而产生不同的训练集/验证集/测试集。在使用留出法时，通常采用多次随机划分，并取平均值作为留出法的评估结果。

交叉验证法

S 折交叉验证法 (cross validation)：数据随机划分为 S 个互不相交且大小相同的子集，利用 $S-1$ 个子集数据训练模型，利用余下的一个子集测试模型（一共有 $C_S^{s-1} = S$ 种组合）。对 S 种组合依次重复进行，获取测试误差的均值。将这个均值作为泛化误差的估计。

将数据集划分为 S 个子集同样存在多种划分方式。S 折交叉验证通常需要随机使用不同的划分重复 p 次，最终的测试误差均值是这 p 次 S 折交叉验证的测试误差均值。

留一法

留一法 (Leave-One-Out，LOO) 是 $S = N$ 时的 S 折交叉验证的一个特例，其中 N 为初始数据集的大小。由于训练集与初始数据集只少了一个样本，因此训练出来的模型与真实模型比较近似。因此留一法评估的结果往往比较准确。

留一法的缺点是：在数据集比较大时计算量太大。比如数据集为一千万个样本，则留一法需要训练一千万个模型（相比较 S 折交叉，假设 $S = 100$，$p = 100$ ，则只需要训练一万个模型）。

自助法

自助法 (bootstrapping) 以自助采样法 (bootstrap sampling) 为基础。

给定包含 N 个样本的原始数据集 T，自助采样法是这样进行的：先从 T 中随机取出一个样本放入采样集 T_s 中，再把该样本放回 T 中（有放回的重复独立采样）。经过 N 次随机采样操作，得到包含 N 个样本的采样集 T_s。

注意：数据集 T 中可能有的样本在采样集 T_s 中出现多次，但是 T 中也可能有样本在 T_s 中从未出现。一个样本始终不在采样集中出现的概率是 $(1-\frac{1}{N})^N$ 。根据：

$$\lim_{N\to\infty}(1-\frac{1}{N})^N = \frac{1}{e} \simeq= 0.368$$

因此 T 中约有 63.2% 的样本出现在了 T_s 中。我们将 T_s 用作训练集，$T - T_s$ 用作测试集。这样的测试称为包外估计（out-of-bag estimate）。

自助法在数据集较小时比较好用。它能从初始数据集中产生多个不同的训练集，这对集成学习等方法有很大的吸引力。自助法产生的数据集改变了初始数据集的分布，从而引入估计偏差。因此在初始数据量足够时，留出法和交叉验证法更为常用。

12.2.4 性能度量

测试准确率与测试错误率

用于度量学习器的预测性能指标如下。

- 测试准确率：测试数据集上的准确率

$$r_{test} = \frac{1}{N'} \sum_{i=1}^{N'} I(y_i = \hat{f}(\vec{\boldsymbol{x}}_i))$$

其中 I 为示性函数。准确率衡量的是有多少比例的样本被正确判别。

- 测试错误率：测试数据集上的错误率

$$e_{test} = \frac{1}{N'} \sum_{i=1}^{N'} I(y_i \neq \hat{f}(\vec{\boldsymbol{x}}_i))$$

其中 I 为示性函数。错误率衡量的是有多少比例的样本被判别错误。

混淆矩阵

对于二类分类问题，通常将关注的类分为正类，其他类分为负类。分类器在测试集上的预测或者正确或者不正确，令

- TP：分类器将正类预测为正类的数量 (True Positive)。
- FN：分类器将正类预测为负类的数量 (False Negative)。
- FP：分类器将负类预测为正类的数量 (False Positive)。
- TN：分类器将负类预测为负类的数量 (True Negative)。

分类结果的混淆矩阵 (confusion matrix) 定义如下：

真实情况	**预测结果**	
	正例	反例
正例	TP(真正例）	FN(假反例）
反例	FP(假正例）	TN(真反例）

定义

- 查准率 (precision) 为：$P = \frac{TP}{TP+FP}$，即所有预测为正类的结果中，真正的正类的比例。
- 查全率 (recall) 为：$R = \frac{TP}{TP+FN}$，即真正的正类中，被分类器找出来的比例。

不同的问题中，判别标准不同。对于推荐系统，更侧重于查准率（即推荐的结果中，用户真正感兴趣的比例）；对于医学诊断系统，更侧重于查全率（即疾病被发现的比例）。

查准率和查全率是一对矛盾的度量。通常查准率高时，查全率往往偏低；而查全率高时，查准率往往偏低。如果希望将所有的正例都找出来（查全率高），最简单的就是将所有

的样本都视为正类，此时有FN=0。此时查准率就偏低（准确性降低）。如果希望查准率高，则可以只挑选有把握的正例。最简单的就是挑选最有把握的那一个样本。此时有FP=0，查全率就偏低（只挑出了一个正例）。

P-R 曲线

我们根据分类器的预测结果对样例进行排序：排在最前面的是分类器认为“最可能”是正例的样本，排在最后面的是分类器认为“最不可能”是正例的样本。假设排序后的样本集合为 $(\vec{x}_1, y_1), (\vec{x}_2, y_2), \cdots, (\vec{x}_N, y_N)$。

根据此顺序，从前到后依次将样本作为正类负类的分界来进行预测。假设第 i 轮，挑选到了样本 $(\vec{x}_i, y_i)$，直接将 $\vec{x}_i$ 记作正类。然后统计 $\vec{x}_1, \vec{x}_2, \cdots, \vec{x}_i$ 全部判定为正类，$\vec{x}_{i+1}, \cdots, \vec{x}_N$ 全部判定为负类时的查全率 R_i、查准率 P_i。

以查准率为纵轴、查全率为横轴作图，就得到查准率-查全率曲线（该曲线由点 $\{(R_1, P_1), (R_2, P_2), \cdots, (R_N, P_N)\}$ 组成）。简称：P-R曲线，显示该曲线的图称为P-R图，如图 12.1 所示。

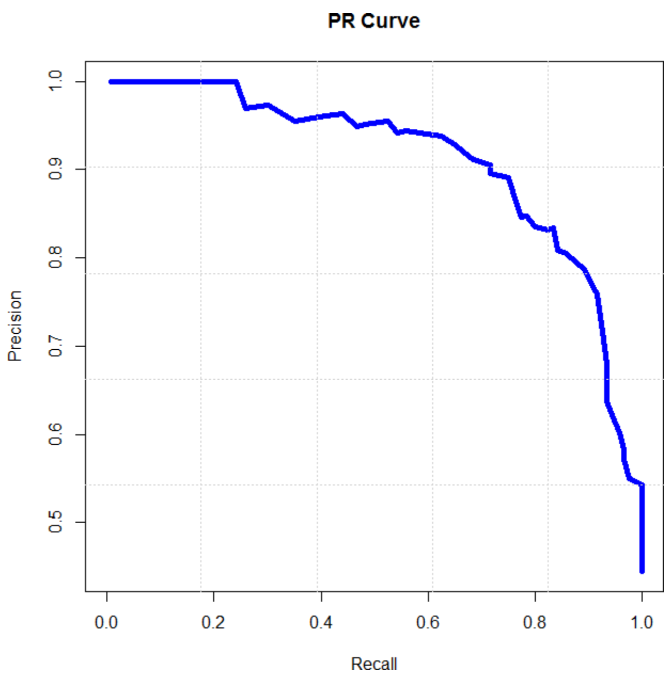


图 12.1 P_R

> 开始时第一个样本（最可能为正例的）预测为正类，其他样本都预测为负类。此时查准率很高（几乎为 1）；查全率很低（几乎为 0，大量的正例没有找到）。结束时所有的样本都预测为正例，此时查全率很高（正例全部找到了，查全率为 1）；查准率很低（大量的负类被预测为正类了）。

P-R图可以用于比较两个分类器的优劣。在进行比较时：

- 如果一个分类器A的P-R曲线被另一个分类器B的曲线完全包围，则B的性能好于A；
- 如果一个分类器A的P-R曲线与另一个分类器B的曲线发生了交叉，则只能在具体的查准率和查全率下进行比较了。

在P-R曲线交叉的情况下，可以考察平衡点。平衡点Break-Even Point:BEP是P-R曲线上查准率等于查全率的点，通常我们认为平衡点较远的P-R曲线较好，如图 12.2 所示。

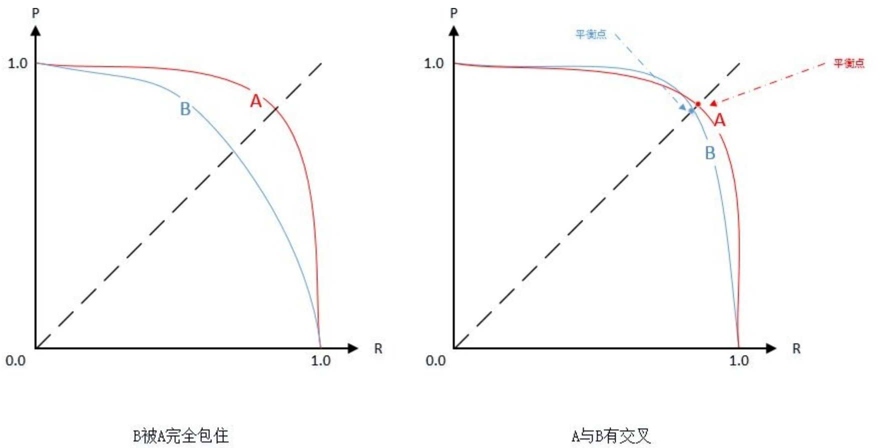


图 12.2 P_R_AB

F_β

定义查准率与查全率的调和均值：$\frac{2}{F_1} = \frac{1}{P} + \frac{1}{R}$。

F_1 更一般的形式：$\frac{1}{F_\beta} = \frac{1}{(1+\beta^2)\times P} + \frac{\beta^2}{(1+\beta^2)\times R}$，其中 $\beta > 0$ 度量了查全率对查准率的相对重要性。

多类混淆矩阵

有时候我们可能得到了多个二分类混淆矩阵（如在多个数据集上进行训练/测试），希望在 m 个二类分类混淆矩阵上综合考察查准率和查全率。

一种方法是：先在各个混淆矩阵上分别计算查准率和查全率，记作：$(P_1, R_1), (P_2, R_2), \cdots, (P_m, R_m)$，然后计算平均值，这样得到的是宏查准率（macro-P），宏查全率（macro-F），宏F1(macro-F1)：

$$\text{macro-P} = \frac{1}{m}\sum_{i=1}^{m} P_i$$

$$\text{macro-R} = \frac{1}{m}\sum_{i=1}^{m} R_i$$

$$\frac{2}{\text{macro-F}_1} = \frac{1}{\text{macro-P}} + \frac{1}{\text{macro-R}}$$

另一种方法是：先将一个混淆矩阵对应元素进行平均，得到 TP, FP, TN, FN 的平均值，记作 $\overline{TP}, \overline{FP}, \overline{TN}, \overline{FN}$，再基于这些平均值计算微查准率 (micro-P)，微查全率 (micro-F)，微F1(micro-F1)：

$$\text{micro-P} = \frac{\overline{TP}}{\overline{TP} + \overline{FP}}$$

$$\text{micro-R} = \frac{\overline{TP}}{\overline{TP} + \overline{FN}}$$

$$\frac{2}{\text{micro-F}_1} = \frac{1}{\text{micro-P}} + \frac{1}{\text{micro-R}}$$

ROC 曲线

首先根据分类器的预测结果对样例进行排序：排在最前面的是分类器认为“最可能”是正例的样本，排在最后面的是分类器认为“最不可能”是正例的样本。假设排序后的样本集合为 $(\vec{\boldsymbol{x}}_1, y_1), (\vec{\boldsymbol{x}}_2, y_2), \cdots, (\vec{\boldsymbol{x}}_N, y_N)$。

根据此顺序，从前到后依次将样本作为正例进行预测。假设第 i 轮，挑选到了样本 $(\vec{\boldsymbol{x}}_i, y_i)$。直接将 $\vec{\boldsymbol{x}}_i$ 记作正类与负类的分隔。然后统计 $\vec{\boldsymbol{x}}_1, \vec{\boldsymbol{x}}_2, \cdots, \vec{\boldsymbol{x}}_i$ 全部判定为正类，$\vec{\boldsymbol{x}}_{i+1}, \cdots, \vec{\boldsymbol{x}}_N$ 全部判定为负类时的真正例率 (True Positive Rate，TPR)、假正例率 (False Positive Rate，FPR) 为：

$$TPR = \frac{TP}{TP + FN} \quad FPR = \frac{FP}{TN + FP}$$

> TPR也就是查全率；FPR刻画的是分类器错认为正类的负实例占所有负实例的比例。
> 当第一个样本为正例，其他样本都为负例时，查全率很低（几乎为 0，因为大量的正例未预测到,TP=1）；FPR为 0（因为此时几乎没有错认的正例，FP=0）。
> 当所有样本为正例时，查全率很高（为 1，因为所有样本都预测为正例了，FN =0）；FPR很高（为 1，因为错认的正例就是所有的负类，这是由于没有预测为负类而导致TN=0）。

以真正例率为纵轴、假正例率为横轴作图，就得到ROC曲线（该曲线由点 $\{(TPR_1, FPR_1), (TPR_2, FPR_2), \cdots, (RPR_N, FPR_N)\}$ 组成）。简称：ROC（Receiver Operating Characteristic）

曲线，显示该曲线的图称为ROC图，如图 12.3 所示。

在ROC图中，对角线对应于随机猜想模型。点(0,1)对应于理想模型（没有预测错误，FPR恒等于 0，TPR恒等于 1）。通常ROC曲线越靠近点(0,1)越好。

- ❑ 如果一个分类器A的ROC曲线被另一个分类器B的曲线完全包住，则可断言：B的性能好于A。
- ❑ 如果一个分类器A的ROC曲线与另一个分类器B的曲线发生了交叉。此时比较ROC曲线下面积的大小，这个面积称为AUC（Area Under ROC Curve）。

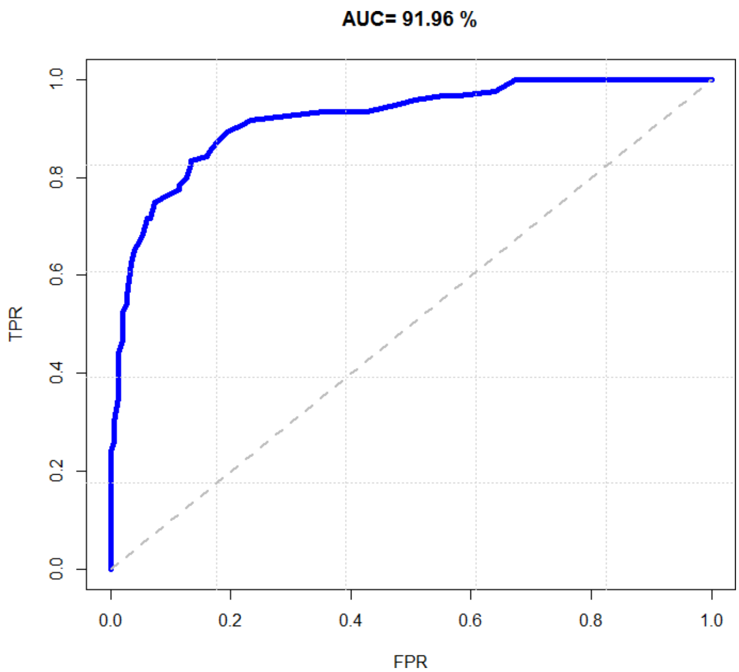


图 12.3　ROC

P-R曲线和ROC曲线刻画的都是阈值对于分类性能的影响：通常一个分类器对样本预测的结果是一个概率结果，比如 0.7。但是样本是不是正类还需要与阈值比较。这个阈值究竟是多少，比如究竟是 0.5 还是 0.9，则影响了分类器的分类性能。

- ❑ 如果我们更重视查准率，则将阈值提升，比如为 0.9。
- ❑ 如果我们更看重查全率，则将阈值下降，比如为 0.5。

P-R曲线和ROC曲线刻画的是随着阈值变化时，查准率、查全率、假正例率等的变化。每一步中，我们直接将 $\vec{x}_i$ 记作正类与负类的分隔，假设分类器对 $\vec{x}_i$ 预测为正例的概率为 p，则实际上该正例作为分隔对应着设置正例阈值为 p。迭代过程对应着正例阈值的下降过程（因为所有实例都以降序排列）。

12.2.5 偏差方差分解

偏差方差分解解释了学习算法的期望泛化误差。我们要刻画算法本质能力，因此要抛开训练集的影响，以回归任务为例。

- 假设在训练集为 T 上学习到的模型为 $f_T(\vec{\boldsymbol{x}};T)$ 。
- 假设训练集 T 中的样本 $\vec{\boldsymbol{x}}$ 的观测值为 y_T ，其真实值为 y。其中 $y_T=y+\epsilon$，ϵ 为观测误差，$E_T(\epsilon)=0$。
- 定义期望预测为 $\bar{f}(\vec{\boldsymbol{x}})=E_T[f_T(\vec{\boldsymbol{x}};T)]$。
- 定义预测方差为 $var(\vec{\boldsymbol{x}})=E_T\left[(f_T(\vec{\boldsymbol{x}};T)-\bar{f}(\vec{\boldsymbol{x}}))^2\right]$。
- 定义噪声方差为 $var(\epsilon)=E_T\left[(y_T-y)^2\right]$。
- 定义偏差bias为 $bias^2(\vec{\boldsymbol{x}})=(\bar{f}(\vec{\boldsymbol{x}})-y)^2$。它刻画了期望输出与真实值之间的差别，则算法的期望泛化误差为：

$$
\begin{aligned}
&E_T\left[(f_T(\vec{\boldsymbol{x}})-y_T)^2\right]\\
&=E_T\left[(f_T(\vec{\boldsymbol{x}})-\bar{f}(\vec{\boldsymbol{x}}))^2\right]+(\bar{f}(\vec{\boldsymbol{x}})-y)^2+E_T\left[(y_T-y)^2\right]\\
&=var(\vec{\boldsymbol{x}})+bias^2(\vec{\boldsymbol{x}})+var(\epsilon)
\end{aligned}
$$

于是泛化误差可以分解为偏差、方差和噪声之和。

- 偏差 $bias^2(\vec{\boldsymbol{x}})$ ：度量了学习算法的期望预测与真实结果之间的偏离程度，刻画了学习算法本身的拟合能力。
- 方差 $var(\vec{\boldsymbol{x}})$：度量了训练集的变动所导致的学习性能的变化，刻画了数据扰动造成的影响。
- 噪声 $var(\epsilon)$ ：度量了在当前任务上任何学习算法所能达到的期望泛化误差的下界，刻画了学习问题本身的难度。

偏差-方差分解表明：泛化性能是由学习算法的能力、数据的充分性以及学习任务本身的难度共同决定的。

通常偏差和方差是有冲突的，这称为偏差-方差困境（`bias-variance dilemma`）。

给定学习任务：

- 当训练不足时，学习器的拟合能力不够强，训练数据的扰动不足以使学习器产生显著变化，此时泛化错误率中偏差占主导；
- 随着训练程度的加深，学习器的拟合能力逐渐增强，此时泛化错误率中偏方差逐渐占主导。

12.3 Python 实践

12.3.1 损失函数

0-1 损失函数

scikit-learn提供了zero_one_loss函数来实现0-1损失函数。其原型为：

```
sklearn.metrics.zero_one_loss(y_true, y_pred, normalize=True, sample_weight=None)
```

参数

- y_true：样本集中每个样本对应的真实值。
- y_pred：学习器对样本集中每个样本的预测的预测值。
- normalize：如果为True，则返回误分类样本的比例；否则返回误分类样本的数量。
- sample_weight：样本权重，默认每个样本的权重为 1。

返回值：0-1损失函数值。

示例：

```
from sklearn.metrics import zero_one_loss
y_true=[1,1,1,1,1,0,0,0,0,0]
y_pred=[0,0,0,1,1,1,1,1,0,0]
print("zero_one_loss<fraction>:",zero_one_loss(y_true,y_pred,normalize=True))
print("zero_one_loss<num>:",zero_one_loss(y_true,y_pred,normalize=False))
```

运行结果如下：

```
zero_one_loss<fraction>: 0.6
zero_one_loss<num>: 6
```

我们将y_pred设置成与y_true有 6 个位置不同。可以看到，当normalize=True时，zero_one_loss 函数返回的是预测错误的样本的比例（6/10=0.6）；当normalize=False时，zero_one_loss函数返回的是预测错误样本数量（即 6）。

对数损失函数

scikit-learn提供了log_loss函数来实现0-1损失函数。其原型为：

```
sklearn.metrics.log_loss(y_true, y_pred, eps=1e-15, normalize=True,
        sample_weight=None)
```

参数

- y_true：样本集中每个样本对应的真实类别。

- y_pred：学习器对样本集中每个样本的预测为每个类别的概率，形状为 n_samples×n_classes。
- eps：当对数的底数为 0 或者 1 时，对数没有定义或者太小。因此需要eps给出此时的对数损失函数值。
- normalize：如果为True，则返回所有样本的对数损失的均值；否则返回所有样本的对数损失的总和。
- sample_weight：样本权重，默认每个样本的权重为 1。

返回值：对数损失函数值。

示例：

```
from sklearn.metrics import log_loss
y_true=[1, 1, 1, 0, 0, 0]
y_pred=[[0.1, 0.9],
        [0.2, 0.8],
        [0.3, 0.7],
        [0.7, 0.3],
        [0.8, 0.2],
        [0.9, 0.1]]
print("log_loss<average>:",log_loss(y_true,y_pred,normalize=True))
print("log_loss<total>:",log_loss(y_true,y_pred,normalize=False))
```

这里有六个样本，其真实分类标记为：1,1,1,0,0,0。给出其预测概率如下。

- 第一个样本：被预测为类别0的概率为 10%，被预测为类别1的概率为 90%。
- 第二个样本：被预测为类别0的概率为 20%，被预测为类别1的概率为 80%。
- 第三个样本：被预测为类别0的概率为 30%，被预测为类别1的概率为 70%。
- 第四个样本：被预测为类别0的概率为 70%，被预测为类别1的概率为 30%。
- 第五个样本：被预测为类别0的概率为 80%，被预测为类别1的概率为 20%。
- 第六个样本：被预测为类别0的概率为 90%，被预测为类别1的概率为 10%。

运行结果如下：

```
log_loss<average>: 0.228393003637
log_loss<total>: 1.37035802182
```

可以看到，当normalize=True时，log_loss函数返回的是返回所有样本的对数损失的均值，这里为 0.228393003637；当normalize=False时，log_loss函数返回所有样本的对数损失的总和，这里为 1.37035802182。

12.3.2 数据集切分

train_test_split

scikit-learn提供的train_test_split函数能够将数据集切分成训练集和测试集两类。函数原型为

```
sklearn.model_selection.train_test_split(*arrays, **options)
```

参数

- *arrays：一个或者多个数据集。
- test_size：一个浮点数，整数或者None，指定测试集的大小。
  - 浮点数：必须是 0.0 到 1.0 之间的数，代表测试集占原始数据集的比例。
  - 整数：代表测试集大小。
  - None：代表测试集大小就是原始数据集大小减去训练集大小。如果训练集大小也指定为None，则test_size设为 0.25。
- train_size：一个浮点数，整数或者None，指定训练集大小。
  - 浮点数：必须是 0.0 到 1.0 之间的数，代表训练集占原始数据集的比例。
  - 整数：代表训练集大小。
  - None：代表训练集大小就是原始数据集大小减去测试集大小。
- random_state：一个整数，或者一个RandomState实例，或者None。
  - 如果为整数，则它指定了随机数生成器的种子。
  - 如果为RandomState实例，则指定了随机数生成器。
  - 如果为None，则使用默认的随机数生成器。
- stratify：一个数组对象或者None。如果它不是None，则原始数据会分层采样，采样的标记数组就由该参数指定。

返回值：为一个列表，依次给出一个或者多个数据集的划分的结果。每个数据集都划分为两部分：训练集、测试集。

示例：

```
from sklearn.model_selection import train_test_split
X=[[1,2,3,4],
   [11,12,13,14],
   [21,22,23,24],
   [31,32,33,34],
   [41,42,43,44],
   [51,52,53,54],
   [61,62,63,64],
   [71,72,73,74]]
y=[1,1,0,0,1,1,0,0]
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y,test_size=0.4, random_state=0)
print("X_train=",X_train)
print("X_test=",X_test)
print("y_train=",y_train)
print("y_test=",y_test)
X_train, X_test, y_train, y_test = train_test_split(X, y,test_size=0.4,
        random_state=0,stratify=y)
print("Stratify:X_train=",X_train)
print("Stratify:X_test=",X_test)
print("Stratify:y_train=",y_train)
print("Stratify:y_test=",y_test)
```

这里给出 8 个样本，其类别标记分别是 `1,1,0,0,1,1,0,0`。分别采用非分层采样和分层采样，运行结果如下：

```
X_train= [[31, 32, 33, 34], [1, 2, 3, 4], [51, 52, 53, 54], [41, 42, 43, 44]]
X_test= [[61, 62, 63, 64], [21, 22, 23, 24], [11, 12, 13, 14], [71, 72, 73, 74]]
y_train= [0, 1, 1, 1]
y_test= [0, 0, 1, 0]
Stratify:X_train= [[41, 42, 43, 44], [61, 62, 63, 64], [1, 2, 3, 4], [71, 72, 73, 74]]
Stratify:X_test= [[21, 22, 23, 24], [31, 32, 33, 34], [11, 12, 13, 14], [51, 52, 53, 54]]
Stratify:y_train= [1, 0, 1, 0]
Stratify:y_test= [0, 0, 1, 1]
```

可以看到如下。

- ❑ 对于非分层采样：训练集中的样本有三个属于类别 1，一个属于类别 0；测试集中的样本有三个属于类别 0，一个属于类别 1；
- ❑ 对于分层采样：训练集中的样本有两个属于类别 1，两个属于类别 0；测试集中的样本有两个属于类别 0，两个属于类别 1。

因此分层采样保证了训练集和测试集中各类别样本的比例与原始数据集一致。

另外关于`test_size`参数，可以看到根据参数的说明，测试集中应该是 $8 \times 0.4 = 3.2$ 个样本。但是这里会适当调整成 4 个样本。

KFold

`scikit-learn`提供的`KFold`类实现了数据集的 k 折交叉切分，它是一个生成器。其原型为

`class sklearn.model_selection.KFold(n_splits=3, shuffle=False, random_state=None)`。

参数

- ❑ `n_folds`：一个整数，即 k（要求该整数值大于等于 2）。

- shuffle：一个布尔值。如果为True，则在切分数据集之前先混洗数据集。
- random_state：一个整数，或者一个RandomState实例，或者None。
  - 如果为整数，则它指定了随机数生成器的种子。
  - 如果为RandomState实例，则指定了随机数生成器。
  - 如果为None，则使用默认的随机数生成器。

方法

- get_n_splits([X, y, groups])：这几个参数都被忽略，用于保持接口的兼容性。它返回的是n_splits参数。
- split(X[, y, groups])：X为训练数据集，形状为(n_samples,n_features)，y为标记信息，形状为(n_samples,)。它切分数据集为训练集和测试集。

KFold首先将 $0\sim(n-1)$ 之间的整数从前到后均匀划分成 n_folds份，每次迭代时依次挑选一份作为测试集的下标。

> 这里并没有随机挑选一份，而是按顺序挑选。如果想随机挑选，则可以在划分之前混洗数据，即shuffle=True。

示例

```
from sklearn.model_selection import KFold
import  numpy as np
X=np.array([[1,2,3,4],
   [11,12,13,14],
   [21,22,23,24],
   [31,32,33,34],
   [41,42,43,44],
   [51,52,53,54],
   [61,62,63,64],
   [71,72,73,74],
   [81,82,83,84]])
y=np.array([1,1,0,0,1,1,0,0,1])

folder=KFold(n_folds=3,random_state=0,shuffle=False)
for train_index,test_index in folder.split(X,y):
      print("Train Index:",train_index)
      print("Test Index:",test_index)
      print("X_train:",X[train_index])
      print("X_test:",X[test_index])
      print("")

shuffle_folder=KFold(n_folds=3,random_state=0,shuffle=True)
for train_index,test_index in shuffle_folder.split(X,y):
      print("Shuffled Train Index:",train_index)
```

```
    print("Shuffled Test Index:",test_index)
    print("Shuffled X_train:",X[train_index])
    print("Shuffled X_test:",X[test_index])
    print("")
```

这里给出了 9 个样本。我们采用了 3 折交叉划分，其中分别给出了不混洗数据和混洗数据的结果。运行结果如下：

```
Train Index: [3 4 5 6 7 8]
Test Index: [0 1 2]
X_train:
[[31 32 33 34]
 [41 42 43 44]
 [51 52 53 54]
 [61 62 63 64]
 [71 72 73 74]
 [81 82 83 84]]
X_test:
[[ 1  2  3  4]
 [11 12 13 14]
 [21 22 23 24]]

Train Index: [0 1 2 6 7 8]
Test Index: [3 4 5]
X_train:
[[ 1  2  3  4]
 [11 12 13 14]
 [21 22 23 24]
 [61 62 63 64]
 [71 72 73 74]
 [81 82 83 84]]
X_test:
[[31 32 33 34]
 [41 42 43 44]
 [51 52 53 54]]

Train Index: [0 1 2 3 4 5]
Test Index: [6 7 8]
X_train:
[[ 1  2  3  4]
 [11 12 13 14]
 [21 22 23 24]
 [31 32 33 34]
 [41 42 43 44]
 [51 52 53 54]]
X_test:
```

```
[[61 62 63 64]
 [71 72 73 74]
 [81 82 83 84]]

Shuffled Train Index: [0 3 4 5 6 8]
Shuffled Test Index: [1 2 7]
Shuffled X_train:
[[ 1  2  3  4]
 [31 32 33 34]
 [41 42 43 44]
 [51 52 53 54]
 [61 62 63 64]
 [81 82 83 84]]
Shuffled X_test:
[[11 12 13 14]
 [21 22 23 24]
 [71 72 73 74]]

Shuffled Train Index: [0 1 2 3 5 7]
Shuffled Test Index: [4 6 8]
Shuffled X_train:
[[ 1  2  3  4]
 [11 12 13 14]
 [21 22 23 24]
 [31 32 33 34]
 [51 52 53 54]
 [71 72 73 74]]
Shuffled X_test:
[[41 42 43 44]
 [61 62 63 64]
 [81 82 83 84]]

Shuffled Train Index: [1 2 4 6 7 8]
Shuffled Test Index: [0 3 5]
Shuffled X_train:
[[11 12 13 14]
 [21 22 23 24]
 [41 42 43 44]
 [61 62 63 64]
 [71 72 73 74]
 [81 82 83 84]]
Shuffled X_test:
[[ 1  2  3  4]
 [31 32 33 34]
 [51 52 53 54]]
```

可以看到：

- 如果不混洗数据（shuffle=False），则数据集被切分成 3 个部分。迭代的结果为
  - 第一次迭代选取的测试集的样本的下标为 [0,1,2]；
  - 第二次迭代选取的测试集的样本的下标为 [3,4,5]；
  - 第三次迭代选取的测试集的样本的下标为 [6,7,8]。
- 如果混洗数据（shuffle=False），数据集同样被切分成 3 个部分。但是现在每一次迭代中，选取的测试集的样本的下标就是随机选取（当然，每次选取不同的样本）。

StratifiedKFold

scikit-learn提供的StratifiedKFold类实现了数据集的分层采样 k 折交叉切分，它也是一个生成器。其原型为

```
class sklearn.model_selection.StratifiedKFold(n_splits=3, shuffle=False,
        random_state=None)
```

参数

- y：样本集的标记序列，可以是列表、数组或其他array-like对象。
- n：一个整数，表示数据集大小。
- n_folds：一个整数，即 k（要求该整数值大于等于 2）。
- shuffle：一个布尔值。如果为True，则在切分数据集之前先混洗数据集。
- random_state：一个整数，或者一个RandomState实例，或者None。
  - 如果为整数，则它指定了随机数生成器的种子。
  - 如果为RandomState实例，则指定了随机数生成器。
  - 如果为None，则使用默认的随机数生成器。

方法

- get_n_splits([X, y, groups])：这几个参数都被忽略，用于保持接口的兼容性。它返回的是n_splits参数。
- split(X[, y, groups])：X为训练数据集，形状为(n_samples,n_features)，y为标记信息，形状为(n_samples,)。它切分数据集为训练集和测试集。

它的用法类似于KFold，但是StratifiedKFold执行的是分层采样，确保训练集、测试集中各类别样本的比例与原始数据集中相同。

这里给出示例来比较 KFold和 StratifiedKFold的区别：

```
from sklearn.model_selection import KFold,StratifiedKFold
import  numpy as np
X=np.array([[1,2,3,4],
   [11,12,13,14],
```

```
    [21,22,23,24],
    [31,32,33,34],
    [41,42,43,44],
    [51,52,53,54],
    [61,62,63,64],
    [71,72,73,74]])

y=np.array([1,1,0,0,1,1,0,0])

folder=KFold(n_folds=4,random_state=0,shuffle=False)
stratified_folder=StratifiedKFold(n_folds=4,random_state=0,shuffle=False)
for train_index,test_index in folder.split(X,y):
      print("Train Index:",train_index)
      print("Test Index:",test_index)
      print("y_train:",y[train_index])
      print("y_test:",y[test_index])
      print("")

for train_index,test_index in stratified_folder.split(X,y):
      print("Stratified Train Index:",train_index)
      print("Stratified Test Index:",test_index)
      print("Stratified y_train:",y[train_index])
      print("Stratified y_test:",y[test_index])
      print("")
```

我们给出 8 个样本，进行 4 折交叉切分。结果如下：

```
Train Index: [2 3 4 5 6 7]
Test Index: [0 1]
y_train: [0 0 1 1 0 0]
y_test: [1 1]

Train Index: [0 1 4 5 6 7]
Test Index: [2 3]
y_train: [1 1 1 1 0 0]
y_test: [0 0]

Train Index: [0 1 2 3 6 7]
Test Index: [4 5]
y_train: [1 1 0 0 0 0]
y_test: [1 1]

Train Index: [0 1 2 3 4 5]
Test Index: [6 7]
y_train: [1 1 0 0 1 1]
y_test: [0 0]
```

```
Stratified Train Index: [1 3 4 5 6 7]
Stratified Test Index: [0 2]
Stratified y_train: [1 0 1 1 0 0]
Stratified y_test: [1 0]

Stratified Train Index: [0 2 4 5 6 7]
Stratified Test Index: [1 3]
Stratified y_train: [1 0 1 1 0 0]
Stratified y_test: [1 0]

Stratified Train Index: [0 1 2 3 5 7]
Stratified Test Index: [4 6]
Stratified y_train: [1 1 0 0 1 0]
Stratified y_test: [1 0]

Stratified Train Index: [0 1 2 3 4 6]
Stratified Test Index: [5 7]
Stratified y_train: [1 1 0 0 1 0]
Stratified y_test: [1 0]
```

可以看到：

- 如果进行普通交叉切分 (KFold)，则出现了测试集全部都是某个分类的情形；
- 如果进行分层采样交叉切分 (StratifiedKFold)，则确保测试集、训练集中各类样本的比例与原始数据集中的一致；
- 为了保证分层采样，StratifiedKFold迭代得到的测试集样本的下标为 [0 2]、[1 3]、[4 6]、[5 7]，而不再是 [0 1]、[2 3]、[4 5]、[6 7]。

LeaveOneOut

scikit-learn提供 LeaveOneOut实现的是留一法拆分数据集 (简称LOO)，它是个生成器，其原型为：

class sklearn.model_selection.LeaveOneOut(n)。

参数

- n：一个整数，表示数据集大小。

LeaveOneOut的用法很简单，它每次迭代时，测试集依次取样本的下标为 0,1,...(n-1)

示例：

```
from sklearn.model_selection import LeaveOneOut
import  numpy as np
```

```
X=np.array([[1,2,3,4],
   [11,12,13,14],
   [21,22,23,24],
   [31,32,33,34]]
)
y=np.array([1,1,0,0])

lo=LeaveOneOut(len(y))
for train_index,test_index in lo:
      print("Train Index:",train_index)
      print("Test Index:",test_index)
      print("X_train:",X[train_index])
      print("X_test:",X[test_index])
      print("")
```

这里给出 4 个样本。运行结果如下：

```
Train Index: [1 2 3]
Test Index: [0]
X_train:
[[11 12 13 14]
 [21 22 23 24]
 [31 32 33 34]]
X_test: [[1 2 3 4]]

Train Index: [0 2 3]
Test Index: [1]
X_train:
[[ 1  2  3  4]
 [21 22 23 24]
 [31 32 33 34]]
X_test: [[11 12 13 14]]

Train Index: [0 1 3]
Test Index: [2]
X_train:
[[ 1  2  3  4]
 [11 12 13 14]
 [31 32 33 34]]
X_test: [[21 22 23 24]]

Train Index: [0 1 2]
Test Index: [3]
X_train:
[[ 1  2  3  4]
 [11 12 13 14]
```

```
 [21 22 23 24]]
X_test: [[31 32 33 34]]
```

可以看到：每次迭代产生的测试集的样本的下标依次为 [0],[1],[2],[3]。

cross_val_score

scikit-learn提供了一个便利函数cross_val_score，它是在指定数据集上运行指定学习器时，通过 k 折交叉获取的最佳性能。其原型为：

```
sklearn.model_selection.cross_val_score(estimator, X, y=None, scoring=None, cv=None,
        n_jobs=1, verbose=0, fit_params=None, pre_dispatch='2*n_jobs')
```

参数

- estimator：指定的学习器，该学习器必须由.fit方法来进行训练。
- X：数据集中的样本集。
- y：数据集中的标记集。
- scoring：一个字符串，或者可调用对象，或者None。它指定了评分函数，其原型是：scorer(estimator, X, y)。如果为None，则默认采用estimator学习器的.score方法。如果为字符串，可以为下列字符串。
  - 'accuracy'：采用的是metrics.accuracy_score评分函数。
  - 'average_precision'：采用的是metrics.average_precision_score评分函数。
  - f1系列：采用的是metrics.f1_score 评分函数，包括如下。
    - 'f1'：用于二类分类。
    - 'f1_micro' ：micro-averaged。
    - 'f1_macro'：macro-averaged。
    - 'f1_weighted'：weighted average。
    - 'f1_samples' ：by multilabel sample。
  - 'log_loss'：采用的是metrics.log_loss评分函数。
  - 'precision'系列，采用的是metrics.precision_score评分函数，具体形式类似f1系列。
  - 'recall' 系列，采用的是metrics.recall_score评分函数，具体形式类似f1系列。
  - 'roc_auc'：采用的是metrics.roc_auc_score 评分函数。
  - 'adjusted_rand_score'：采用的是metrics.adjusted_rand_score 评分函数。
  - 'mean_absolute_error'：采用的是metrics.mean_absolute_error 评分函数。
  - 'mean_squared_error’'：采用的是metrics.mean_squared_error 评分函数。
  - 'median_absolute_error'：采用的是metrics.median_absolute_error 评分函数。
  - 'r2'：采用的是metrics.r2_score 评分函数。
- cv：一个整数、k 折交叉生成器、一个迭代器、或者None。

  - ❍ 如果为None，则使用默认的 3 折交叉生成器。
  - ❍ 如果为整数，则指定了 k 折交叉生成器的 k 值。
  - ❍ 如果为 k 折交叉生成器，则直接指定了 k 折交叉生成器。
  - ❍ 如果为迭代器，则迭代器的结果就是数据集划分的结果。
- ❑ fit_params：一个字典，指定了estimator执行.fit方法时的关键字参数。
- ❑ n_jobs：并行性。默认为 −1 表示派发任务到所有计算机的 CPU 上。
- ❑ verbose：一个整数，用于控制输出日志。
- ❑ pre_dispatch：一个整数或者字符串，用于控制并行执行时，分发的总的任务数量。

返回值：返回一个浮点数的数组。每个浮点数都是针对某次 k 折交叉的数据集上 estimator 预测性能的得分。

之所以称为便利函数，是因为完全可以凭借现有的函数手动完成这个功能，步骤如下。

- ❑ k 折交叉划分数据集。
- ❑ 对每次划分结果执行：
  - ❍ 在训练集上训练学习器；
  - ❍ 用学习器预测测试集，返回测试性能得分。
- ❑ 收集所有的测试性能得分，放入一个数组并返回。

示例：

```
from sklearn.model_selection import cross_val_score
from sklearn.datasets import  load_digits
from sklearn.svm import  LinearSVC

digits=load_digits()
X=digits.data
y=digits.target

result=cross_val_score(LinearSVC(),X,y,cv=10)
print("Cross Val Score is:",result)
```

使用 10 折交叉划分原始数据集，使用线性支持向量机作为学习器。结果如下：

```
Cross Val Score is: [ 0.9027027   0.95081967  0.87845304  0.9   0.94413408  0.94972067
  0.96648045  0.93820225  0.86440678  0.94318182]
```

cross_val_score依次选择第 0~9折的数据作为测试数据集。可以看到同一个线性支持向量机在这 10 种“训练集--预测集”的组合上的预测性能差距较大，从 0.86440678 到 0.96648045 不等。

12.3.3　性能度量

在scikit-learn中有三种方法来评估模型的预测性能：

- 学习器的.score方法。该方法是每个学习器的方法，在讲解每一类学习器时会给出说明。
- 通过使用cross-validation中的模型评估工具来评估，如cross-validation.cross_val_score等方法。
- 通过scikit-learn的metrics模块中的函数来评估模型的预测性能。这里重点讲解这些函数。

分类问题的性能度量

（1）accuracy_score　scikit-learn提供的accuracy_score函数用于计算分类结果的准确率，其原型为：

```
sklearn.metrics.accuracy_score(y_true, y_pred, normalize=True, sample_weight=None)
```

参数

- y_true：样本集的真实标记集合。
- y_pred：分类器对样本集预测的预测值。
- normalize：如果为True，则返回分类正确的比例（准确率），为一个浮点数；否则返回分类正确的数量，为一个整数。
- sample_weight：样本权重，默认每个样本的权重为 1。

返回值：

- 如果normalize为True，则返回准确率；
- 如果normalize为False，则返回正确分类的数量。

示例：

```
from sklearn.metrics import accuracy_score
y_true=[1,1,1,1,1,0,0,0,0,0]
y_pred=[0,0,1,1,0,0,1,1,0,0]
print('Accuracy Score(normalize=True):',accuracy_score(y_true,y_pred,normalize=True))
print('Accuracy Score(normalize=False):',accuracy_score(y_true,y_pred,normalize=False))
```

这里给出 10 个样本的真实分类标记，以及对应的预测分类标记。执行结果如下：

```
Accuracy Score(normalize=True): 0.5
Accuracy Score(normalize=False): 5
```

可以看到，accuracy_score就是简单地比较y_pred与y_true不同的样本个数。

（2）precision_score scikit-learn提供的precision_score函数用于计算分类结果的查准率，其原型为：

```
sklearn.metrics.precision_score(y_true, y_pred, labels=None, pos_label=1,
        average='binary', sample_weight=None)
```

参数

- y_true：样本集的真实标记集合。
- y_pred：分类器对样本集预测的预测值。
- pos_label：一个字符串或者整数，指定哪个标记值属于正类。
- average：一个字符串，用于多类分类问题。
- sample_weight：样本权重，默认每个样本的权重为 1。

返回值：查准率。即预测结果为正类的那些样本中，有多少比例确实是正类。

示例：

```
from sklearn.metrics import accuracy_score,precision_score
y_true=[1,1,1,1,1,0,0,0,0,0]
y_pred=[0,0,1,1,0,0,0,0,0,0]
print('Accuracy Score:',accuracy_score(y_true,y_pred,normalize=True))
print('Precision Score:',precision_score(y_true,y_pred))
```

这里给出 10 个样本的真实分类标记，以及对应的预测分类标记，而且我们将 1 标记为正类。执行结果如下：

```
Accuracy Score: 0.7
Precision Score: 1.0
```

可以看到，准确率accuracy_score 为 70%，但是查准率 precision_score 为 100%（此处预测为正类的样本中，它们的真实类别标记确实为正类）。

（3）recall_score scikit-learn提供的recall_score函数用于计算分类结果的查全率，其原型为：

```
sklearn.metrics.recall_score(y_true, y_pred, labels=None, pos_label=1,
        average='binary', sample_weight=None)
```

参数

- y_true：样本集的真实标记集合。
- y_pred：分类器对样本集预测的预测值。
- pos_label：一个字符串或者整数，指定哪个标记值属于正类。
- average：一个字符串，用于多类分类问题。
- sample_weight：样本权重，默认每个样本的权重为 1。

返回值：查全率。即真实的正类中，有多少比例被预测为正类。

示例：

```
from sklearn.metrics import accuracy_score,precision_score,recall_score
y_true=[1,1,1,1,1,0,0,0,0,0]
y_pred=[0,0,1,1,0,0,0,0,0,0]
print('Accuracy Score:',accuracy_score(y_true,y_pred,normalize=True))
print('Precision Score:',precision_score(y_true,y_pred))
print('Recall Score:',recall_score(y_true,y_pred))
```

这里给出 10 个样本的真实分类标记，以及对应的预测分类标记，而且将 1 标记为正类。执行结果如下：

```
Accuracy Score: 0.7
Precision Score: 1.0
Recall Score: 0.4
```

可以看到，准确率accuracy_score 为 70%，查准率 precision_score 为 100%，但是查全率为 40%（一共 5 个正类，但是只成功预测其中的两个）。

（4）f1_score　scikit-learn提供的f1_score函数用于计算分类结果的 F_1 值，其原型为：

```
sklearn.metrics.f1_score(y_true, y_pred, labels=None, pos_label=1,
        average='binary', sample_weight=None)
```

参数

- y_true：样本集的真实标记集合。
- y_pred：分类器对样本集预测的预测值。
- pos_label：一个字符串或者整数，指定哪个标记值属于正类。
- average：一个字符串，用于多类分类问题。
- sample_weight：样本权重，默认每个样本的权重为 1。

返回值：F_1 值。即查准率和查全率的调和均值。

示例：

```
from sklearn.metrics import accuracy_score,precision_score,recall_score,f1_score
y_true=[1,1,1,1,1,0,0,0,0,0]
y_pred=[0,0,1,1,0,0,0,0,0,0]
print('Accuracy Score:',accuracy_score(y_true,y_pred,normalize=True))
print('Precision Score:',precision_score(y_true,y_pred))
print('Recall Score:',recall_score(y_true,y_pred))
print('F1 Score:',f1_score(y_true,y_pred))
```

这里给出 10 个样本的真实分类标记，以及对应的预测分类标记，而且我们将 1 标记为正类。执行结果如下：

```
Accuracy Score: 0.7
Precision Score: 1.0
Recall Score: 0.4
F1 Score: 0.571428571429
```

可以看到，准确率accuracy_score 为 70%，查准率 precision_score 为 100% ，查全率为 40% ，F_1 值为 0.571428571429（$\frac{2}{0.571428571429} = \frac{1}{1.0} + \frac{1}{0.4}$）。

（5）fbeta_score scikit-learn提供的fbeta_score函数用于计算分类结果的 F_β 值，其原型为：

```
sklearn.metrics.fbeta_score(y_true, y_pred, beta, labels=None, pos_label=1,
        average='binary', sample_weight=None)
```

参数

- y_true：样本集的真实标记集合。
- y_pred：分类器对样本集预测的预测值。
- beta：β 值。
- pos_label：一个字符串或者整数，指定哪个标记值属于正类。
- average：一个字符串，用于多类分类问题。
- sample_weight：样本权重，默认每个样本的权重为 1。

返回值：F_β 值。

示例：

```
from sklearn.metrics import accuracy_score,precision_score,recall_score,
    f1_score,fbeta_score
y_true=[1,1,1,1,1,0,0,0,0,0]
y_pred=[0,0,1,1,0,0,0,0,0,0]
print('Accuracy Score:',accuracy_score(y_true,y_pred,normalize=True))
print('Precision Score:',precision_score(y_true,y_pred))
print('Recall Score:',recall_score(y_true,y_pred))
print('F1 Score:',f1_score(y_true,y_pred))
print('Fbeta Score(beta=0.001):',fbeta_score(y_true,y_pred,beta=0.001))
print('Fbeta Score(beta=1):',fbeta_score(y_true,y_pred,beta=1))
print('Fbeta Score(beta=10):',fbeta_score(y_true,y_pred,beta=10))
print('Fbeta Score(beta=10000):',fbeta_score(y_true,y_pred,beta=10000))
```

这里给出 10 个样本的真实分类标记，以及对应的预测分类标记，而且将 1 标记为正类。执行结果如下：

```
Accuracy Score: 0.7
Precision Score: 1.0
Recall Score: 0.4
F1 Score: 0.571428571429
```

```
Fbeta Score(beta=0.001): 0.999998500004
Fbeta Score(beta=1): 0.571428571429
Fbeta Score(beta=10): 0.402390438247
Fbeta Score(beta=10000): 0.4000000024
```

可以看到，准确率accuracy_score 为 70%，查准率 precision_score 为 100%，查全率为 40%，F_1 值为 0.571428571429。给出了 β 为 0.001,1,10,10000时的 F_β 的值依次为 0.999998500004, 0.571428571429,0.402390438247,0.4000000024。从中可以发现：

- 当 $\beta \to 0$ 时，$F_\beta \to P$。
- 当 $\beta \to +\infty$ 时，$F_\beta \to R$。
- 当 $\beta = 1$ 时，$F_\beta = F_1$。

（6）classification_report scikit-learn提供的classification_report函数以文本方式给出了分类结果的主要预测性能指标。其原型为：

```
sklearn.metrics.classification_report(y_true, y_pred, labels=None,
        target_names=None, sample_weight=None, digits=2)
```

参数

- y_true：样本集的真实标记集合。
- y_pred：分类器对样本集预测的预测值。
- labels：指定报告中出现哪些类别。
- target_names：指定报告中类别对应显示出来的名字。
- digits：用于格式化报告中的浮点数，保留几位小数。
- sample_weight：样本权重，默认每个样本的权重为 1。

返回值：预测性能指标的字符串。

示例：

```
from sklearn.metrics import classification_report
y_true=[1,1,1,1,1,0,0,0,0,0]
y_pred=[0,0,1,1,0,0,0,0,0,0]
print('Classification Report:\n',classification_report(y_true,y_pred,
            target_names=["class_0","class_1"]))
```

这里给出 10 个样本的真实分类标记，以及对应的预测分类标记。执行结果如下：

```
Classification Report:
              precision    recall  f1-score   support


    class_0       0.62      1.00      0.77         5
    class_1       1.00      0.40      0.57         5
```

```
avg / total       0.81      0.70      0.67        10
```

其中

- precision列：给出了查准率。它依次将类别 0 作为正类，类别 1 作为正类……
- recall列：给出了查全率。它依次将类别 0 作为正类，类别 1 作为正类……
- recall列：给出了 F_1 值。
- support列：给出了该类有多少个样本。
- avg / total行：对于precision,recall,recall，给出了该列数据的算术平均；对于support列，给出了该列的算术和（其实就等于样本集总样本数量）。

（7）confusion_matrix scikit-learn提供的confusion_matrix函数给出了分类结果的混淆矩阵。其原型为：

```
sklearn.metrics.confusion_matrix(y_true, y_pred, labels=None)
```

参数

- y_true：样本集的真实标记集合。
- y_pred：分类器对样本集预测的预测值。
- labels：指定混淆矩阵中出现哪些类别。

返回值：分类结果的混淆矩阵。

示例：

```
from sklearn.metrics import confusion_matrix
y_true=[1,1,1,1,1,0,0,0,0,0]
y_pred=[0,0,1,1,0,0,0,0,0,0]
print('Confusion Matrix:\n',confusion_matrix(y_true,y_pred,labels=[0,1]))
```

这里给出 10 个样本的真实分类标记，以及对应的预测分类标记。执行结果如下：

```
Confusion Matrix:
 [[5 0]
 [3 2]]
```

其中

- 5：表示真实标记为 0，预测标记为0的样本数量为 5。
- 0：表示真实标记为 0，预测标记为1的样本数量为 0。
- 3：表示真实标记为 1，预测标记为0的样本数量为 3。
- 2：表示真实标记为 1，预测标记为1的样本数量为 2。

（8）precision_recall_curve scikit-learn提供的precision_recall_curve函数用于计算分类结果的P-R曲线。其原型为：

```
sklearn.metrics.precision_recall_curve(y_true, probas_pred, pos_label=None,
     sample_weight=None)
```

参数

- y_true：样本集的真实标记集合。
- probas_pred：依次指定每个样本为正类的概率。
- pos_label：正类的类别标记。
- sample_weight：样本权重，默认每个样本的权重为 1。

返回值：一个元组，元组内的元素分别为

- P-R曲线的查准率序列。该序列是递增序列，序列第 i 个元素是当正类的阈值为 thresholds[i] 时的查准率。
- P-R曲线的查全率序列。该序列是递减序列，序列第 i 个元素是当正类的阈值为 thresholds[i] 时的查全率。
- P-R曲线的阈值序列。该序列是一个递增序列，给出了判定为正例时的probas_pred的阈值。

示例：

```
from sklearn.metrics import precision_recall_curve
from sklearn.datasets import load_iris
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import  SVC
from sklearn.model_selection import train_test_split
import  matplotlib.pyplot as plt
from sklearn.preprocessing import label_binarize
import  numpy as np
### 加载数据
iris=load_iris()
X=iris.data
y=iris.target
# 二元化标记
y = label_binarize(y, classes=[0, 1, 2])
n_classes = y.shape[1]
#### 添加噪声
np.random.seed(0)
n_samples, n_features = X.shape
X = np.c_[X, np.random.randn(n_samples, 200 * n_features)]

X_train,X_test,y_train,y_test=train_test_split(X,y,
        test_size=0.5,random_state=0)
```

```
### 训练模型
clf=OneVsRestClassifier(SVC(kernel='linear', probability=True,random_state=0))
clf.fit(X_train,y_train)
y_score = clf.fit(X_train, y_train).decision_function(X_test)
### 获取 P-R
fig=plt.figure()
ax=fig.add_subplot(1,1,1)
precision = dict()
recall = dict()
for i in range(n_classes):
    precision[i], recall[i], _ = precision_recall_curve(y_test[:, i],
                                                        y_score[:, i])
    ax.plot(recall[i],precision[i],label="target=%s"%i)
ax.set_xlabel("Recall Score")
ax.set_ylabel("Precision Score")
ax.set_title("P-R")
ax.legend(loc='best')
ax.set_xlim(0,1.1)
ax.set_ylim(0,1.1)
ax.grid()
plt.show()
```

使用scikit-learn自带的数据集iris，使用OneVsRestClassifier作为分类器。由于添加了噪声从而增加难度。运行结果如图 12.4 所示。

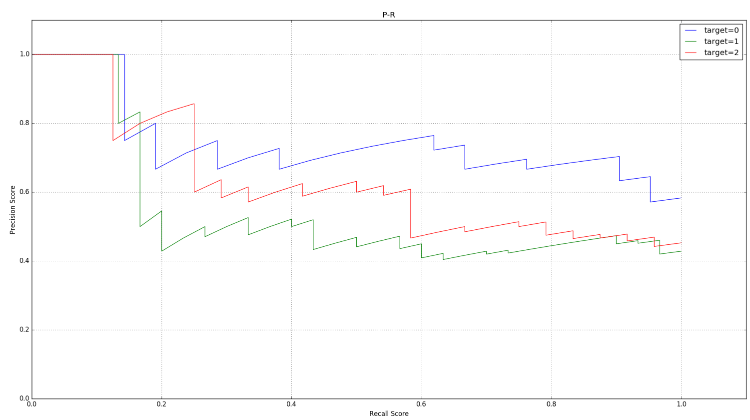


图 12.4 P_R2

（9）roc_curve scikit-learn提供的roc_curve函数用于计算分类结果的ROC曲线。其原型为：

```
sklearn.metrics.roc_curve(y_true, y_score, pos_label=None, sample_weight=None,
    drop_intermediate=True)
```

参数

- y_true：样本集的真实标记集合。
- y_score：依次指定每个样本为正类的概率。
- pos_label：正类的类别标记。
- sample_weight：样本权重，默认每个样本的权重为 1。
- drop_intermediate：一个布尔值。如果为True，则抛弃某些不可能出现在ROC曲线上的阈值。

返回值：一个元组，元组内的元素分别为：

- ROC曲线的 FPR 序列。该序列是递增序列，序列第 i 个元素是当正类的阈值为 thresholds[i] 时的假正例率。
- ROC曲线的 TPR 序列。该序列是递增序列，序列第 i 个元素是当正类的阈值为 thresholds[i] 时的真正例率。
- ROC曲线的阈值序列thresholds。该序列是一个递减序列，给出了判定为正例时的阈值。

示例：

```
from sklearn.metrics import roc_curve,roc_auc_score
from sklearn.datasets import load_iris
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import  SVC
from sklearn.model_selection import train_test_split
import  matplotlib.pyplot as plt
from sklearn.preprocessing import label_binarize
import  numpy as np

### 加载数据
iris=load_iris()
X=iris.data
y=iris.target
# 二元化标记
y = label_binarize(y, classes=[0, 1, 2])
n_classes = y.shape[1]
#### 添加噪声
np.random.seed(0)
n_samples, n_features = X.shape
X = np.c_[X, np.random.randn(n_samples, 200 * n_features)]

X_train,X_test,y_train,y_test=train_test_split(X,y,
        test_size=0.5,random_state=0)
### 训练模型
clf=OneVsRestClassifier(SVC(kernel='linear', probability=True,random_state=0))
clf.fit(X_train,y_train)
```

```
y_score = clf.fit(X_train, y_train).decision_function(X_test)
### 获取 ROC
fig=plt.figure()
ax=fig.add_subplot(1,1,1)
fpr = dict()
tpr = dict()
roc_auc=dict()
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test[:, i],y_score[:, i])
    roc_auc[i] = roc_auc_score(fpr[i], tpr[i])
    ax.plot(fpr[i],tpr[i],label="target=%s,auc=%s"%(i,roc_auc[i]))
ax.plot([0, 1], [0, 1], 'k--')
ax.set_xlabel("FPR")
ax.set_ylabel("TPR")
ax.set_title("ROC")
ax.legend(loc="best")
ax.set_xlim(0,1.1)
ax.set_ylim(0,1.1)
ax.grid()
plt.show()
```

使用scikit-learn自带的数据集iris，使用OneVsRestClassifier作为分类器。由于添加了噪声从而增加难度，运行结果如图 12.5 所示。

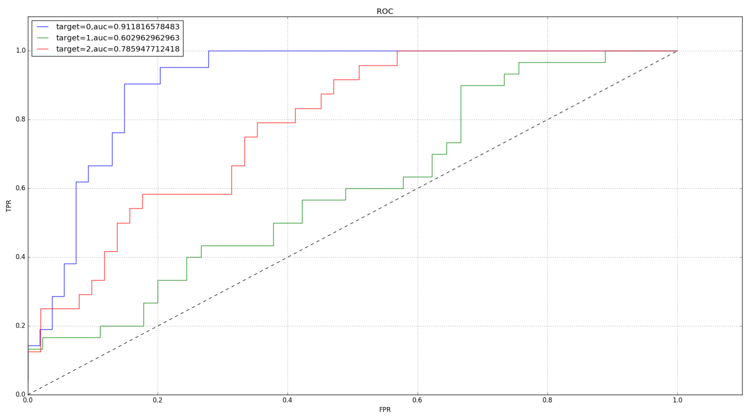


图 12.5　ROC2

（10）**roc_auc_score**　scikit-learn提供的roc_auc_score函数用于计算分类结果的ROC曲线的面积AUC。其原型为：

```
sklearn.metrics.roc_auc_score(y_true, y_score, average='macro', sample_weight=None)
```

参数

- ❏ y_true：样本集的真实标记集合。

- y_score：依次指定每个样本为正类的概率。
- average：一个字符串，用于多类分类问题。
- sample_weight：样本权重，默认每个样本的权重为 1。

返回值：AUC值。

示例参考上面ROC的示例。

回归问题的性能度量

（1）mean_absolute_error scikit-learn提供的mean_absolute_error函数用于计算回归预测误差的绝对值的平均值 (mean absolute error:MAE)，其原型为：

```
sklearn.metrics.mean_absolute_error(y_true, y_pred, sample_weight=None,
        multioutput='uniform_average')
```

参数

- y_true：样本集的真实值集合。
- y_pred：回归器对样本集的预测值。
- multioutput：指定对于多输出变量的回归问题的误差类型。可以为如下。
  - 'raw_values'：对每个输出变量，计算其误差。
  - 'uniform_average'：计算其所有输出变量的误差的平均值。
- sample_weight：样本权重，默认每个样本的权重为 1。

返回值：误差的绝对值的平均值。

示例：

```
from sklearn.metrics import mean_absolute_error

y_true=[1,1,1,1,1,2,2,2,0,0]
y_pred=[0,0,0,1,1,1,0,0,0,0]

print("Mean Absolute Error:",mean_absolute_error(y_true,y_pred))
```

这里给出 10 个样本的真实输出和预测输出。执行结果如下（$0.8 = \frac{1+1+1+1+2+2}{10}$ 为预测误差的绝对值的平均值)。

```
Mean Absolute Error: 0.8
```

（2）mean_squared_error scikit-learn提供的mean_squared_error函数用于计算回归预测误差的平方的平均值 (mean square error:MSE)，其原型为：

```
sklearn.metrics.mean_squared_error(y_true, y_pred, sample_weight=None,
        multioutput='uniform_average')
```

参数

- ❑ y_true：样本集的真实值集合。
- ❑ y_pred：回归器对样本集的预测值。
- ❑ multioutput：指定对于多输出变量的回归问题的误差类型。可以为如下。
  - ❍ 'raw_values'：对每个输出变量，计算其误差。
  - ❍ 'uniform_average'：计算其所有输出变量的误差的平均值。
- ❑ sample_weight：样本权重，默认每个样本的权重为 1。

返回值：误差的平方的平均值。

示例：

```
from sklearn.metrics import mean_squared_error,mean_absolute_error

y_true=[1,1,1,1,1,2,2,2,0,0]
y_pred=[0,0,0,1,1,1,0,0,0,0]

print("Mean Absolute Error:",mean_absolute_error(y_true,y_pred))
print("Mean Square Error:",mean_squared_error(y_true,y_pred))
```

这里给出 10 个样本的真实输出和预测输出。执行结果如下：

```
Mean Absolute Error: 0.8
Mean Square Error: 1.2
```

预测误差的绝对值的平均值为 0.8，预测误差的平方的平均值为 1.2（$1.2=\frac{1+1+1+1+4+4}{10}$ 为预测误差的平方的平均值）。

验证曲线

scikit-learn提供的validation_curve函数给出了学习器因为某个参数的不同取值在同一个测试集上预测的性能曲线。其原型为：

```
sklearn.model_selection.validation_curve(estimator, X, y, param_name, param_range,
        cv=None, scoring=None, n_jobs=1, pre_dispatch='all', verbose=0)
```

参数

- ❑ estimator：一个学习器对象。它必须有.fit方法用于学习，.predict方法用于预测。
- ❑ X：训练样本集。
- ❑ y：训练样本对应的标签集合。
- ❑ param_name：一个字符串，指定了学习器需要变化的参数。
- ❑ param_range：一个序列，指定了param_name指定的参数的取值范围。
- ❑ cv：一个整数、k 折交叉生成器、一个迭代器、或者None。

- ❍ 如果为None，则使用默认的 3 折交叉生成器。
- ❍ 如果为整数，则指定了 k 折交叉生成器的 k 值。
- ❍ 如果为 k 折交叉生成器，则直接指定了 k 折交叉生成器。
- ❍ 如果为迭代器，则迭代器的结果就是数据集划分的结果。

- ❑ scoring：一个字符串，或者可调用对象，或者None。它指定了评分函数，其原型是：scorer(estimator, X, y)。如果为None，则使用estimator的.score方法。如果为字符串，可以为下列字符串。
  - ❍ 'accuracy': 采用的是metrics.accuracy_score评分函数。
  - ❍ 'average_precision'：采用的是metrics.average_precision_score评分函数。
  - ❍ f1系列：采用的是metrics.f1_score 评分函数，包括如下。
    - ❖ 'f1'：用于二类分类。
    - ❖ 'f1_micro' ：micro-averaged。
    - ❖ 'f1_macro'：macro-averaged。
    - ❖ 'f1_weighted'：weighted average。
    - ❖ 'f1_samples' ：by multilabel sample。
  - ❍ 'log_loss'：采用的是metrics.log_loss评分函数。
  - ❍ 'precision'系列，采用的是metrics.precision_score评分函数，具体形式类似f1系列。
  - ❍ 'recall' 系列，采用的是metrics.recall_score评分函数，具体形式类似f1系列。
  - ❍ 'roc_auc'：采用的是metrics.roc_auc_score 评分函数。
  - ❍ 'adjusted_rand_score'：采用的是metrics.adjusted_rand_score 评分函数。
  - ❍ 'mean_absolute_error'：采用的是metrics.mean_absolute_error 评分函数。
  - ❍ 'mean_squared_error’'：采用的是metrics.mean_squared_error 评分函数。
  - ❍ 'median_absolute_error'：采用的是metrics.median_absolute_error 评分函数。
  - ❍ 'r2'：采用的是metrics.r2_score 评分函数。
- ❑ verbose：一个整数。它控制输出日志的内容。该值越大，输出越多的内容。
- ❑ n_jobs：并行性。默认为 −1 表示派发任务到所有计算机的 CPU 上。
- ❑ pre_dispatch：一个整数或者字符串，用于控制并行执行时，分发的总的任务数量。

返回值：返回一个元组，其元素依次为如下。

- ❑ train_scores：学习器在训练集上的预测得分的序列（针对不同的参数值），是个二维数组。
- ❑ test_scores：学习器在测试集上的预测得分的序列（针对不同的参数值），是个二维数组。

> 因为对于每个固定的参数值，k 折交叉会导致产生若干个不同的集合划分，产生多个预测得分。

示例：

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import load_digits
from sklearn.svm import LinearSVC
from sklearn.model_selection import validation_curve

### 加载数据
digits = load_digits()
X,y=digits.data,digits.target
#### 获取验证曲线 ######
param_name="C"
param_range = np.logspace(-2, 2)
train_scores, test_scores = validation_curve(LinearSVC(), X, y, param_name=param_name,
         param_range=param_range,cv=10, scoring="accuracy")
###### 对每个 C ，获取 10 折交叉上的预测得分上的均值和方差 #####
train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)
####### 绘图 ######
fig=plt.figure()
ax=fig.add_subplot(1,1,1)

ax.semilogx(param_range, train_scores_mean, label="Training Accuracy", color="r")
ax.fill_between(param_range, train_scores_mean - train_scores_std,
                 train_scores_mean + train_scores_std, alpha=0.2, color="r")
ax.semilogx(param_range, test_scores_mean, label="Testing Accuracy", color="g")
ax.fill_between(param_range, test_scores_mean - test_scores_std,
                 test_scores_mean + test_scores_std, alpha=0.2, color="g")

ax.set_title("Validation Curve with LinearSVC")
ax.set_xlabel("C")
ax.set_ylabel("Score")
ax.set_ylim(0,1.1)
ax.legend(loc='best')
plt.show()
```

这里使用的数据集是 scikit-learn 自带的 digits 数据集。使用线性支持向量机作为分类器，待变化的参数为C‘参数，其变化范围为 0.01 ~ 100。使用准确率作为评分标准，且使用 10 折交叉进行多次训练。

由于采用了 10 折交叉，因此对于每一个 C的值，其训练得分和预测得分均为 10 个。为了绘制C的性能变化曲线，使用了这 10 个得分的均值和标准差。分别绘制训练得分和预测得分的 10 折交叉的均值随C的变化曲线，同时在其上叠加标准差来绘制填充带，运行结果如图 12.6 所示。

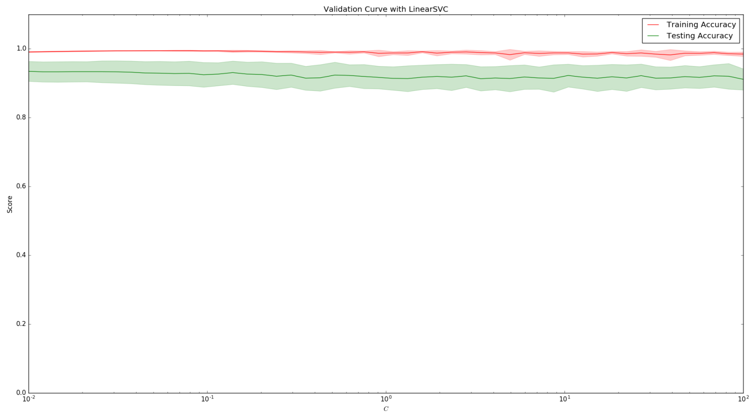


图 12.6 validation_curve

学习曲线

scikit-learn提供的learning_curve函数给出了学习器因为数据集大小的不同而导致的学习器在训练集和测试集上预测的性能曲线。其原型为：

```
sklearn.model_selection.learning_curve(estimator, X, y, train_sizes=array(
    [ 0.1, 0.33, 0.55, 0.78, 1. ]), cv=None, scoring=None,
     exploit_incremental_learning=False, n_jobs=1, pre_dispatch='all', verbose=0
```

该函数的作用是：评估随着样本集的大小的变化对学习器的性能的影响。每次从原始数据集中抽取不同数量的部分数据来训练和预测，抽取出来的数据组成的集合为考察数据集。

参数

- estimator：一个学习器对象。它必须有.fit方法用于学习，.predict方法用于预测。
- X：训练样本集。
- y：训练样本对应的标签集合。
- train_sizes：一个数组，指定了考察数据集的大小。
    - 如果元素类型为浮点数，则数组的元素就是相对原始数据集的比例（不超过 1.0）。
    - 如果元素类型为整数，则数组元素就是考察数据集的绝对大小（不超过原始数据集的大小）。
- cv：一个整数、k 折交叉生成器、一个迭代器、或者None。
    - 如果为None，则使用默认的 3 折交叉生成器。
    - 如果为整数，则指定了 k 折交叉生成器的 k 值。

- 如果为 k 折交叉生成器，则直接指定了 k 折交叉生成器。
- 如果为迭代器，则迭代器的结果就是数据集划分的结果。

- scoring：一个字符串，或者可调用对象，或者None。它指定了评分函数，其原型是：scorer(estimator, X, y)。如果为None，则使用estimator的.score方法。如果为字符串，可以为下列字符串。
  - 'accuracy': 采用的是metrics.accuracy_score评分函数。
  - 'average_precision'：采用的是metrics.average_precision_score评分函数。
  - f1系列：采用的是metrics.f1_score 评分函数，包括如下。
    - 'f1'：用于二类分类。
    - 'f1_micro' ：micro-averaged。
    - 'f1_macro'：macro-averaged。
    - 'f1_weighted'：weighted average。
    - 'f1_samples' ：by multilabel sample。
  - 'log_loss'：采用的是metrics.log_loss评分函数。
  - 'precision'系列，采用的是metrics.precision_score评分函数，具体形式类似f1系列。
  - 'recall' 系列，采用的是metrics.recall_score评分函数，具体形式类似f1系列。
  - 'roc_auc'：采用的是metrics.roc_auc_score 评分函数。
  - 'adjusted_rand_score'：采用的是metrics.adjusted_rand_score 评分函数。
  - 'mean_absolute_error'：采用的是metrics.mean_absolute_error 评分函数。
  - 'mean_squared_error’'：采用的是metrics.mean_squared_error 评分函数。
  - 'median_absolute_error'：采用的是metrics.median_absolute_error 评分函数。
  - 'r2'：采用的是metrics.r2_score 评分函数。
- verbose：一个整数。它控制输出日志的内容。该值越大，输出越多的内容。
- n_jobs：并行性。默认为 −1 表示派发任务到所有计算机的 CPU 上。
- pre_dispatch：一个整数或者字符串，用于控制并行执行时分发的总的任务数量。

返回值：返回一个元组，其元素依次为如下。

- train_sizes_abs：考察数据集大小组成的序列。
- train_scores：学习器在训练集上的预测得分的序列（针对不同的考察数据集），是个二维数组。
- test_scores：学习器在测试集上的预测得分的序列（针对不同的考察数据集），是个二维数组。

> 因为对于每个固定的考察数据集，k 折交叉会导致产生若干个不同的集合划分，产生多个预测得分。

示例：

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import load_digits
from sklearn.svm import LinearSVC
from sklearn.model_selection import learning_curve

### 加载数据
digits = load_digits()
X,y=digits.data,digits.target
#### 获取学习曲线 ######
train_sizes=np.linspace(0.1,1.0,endpoint=True,dtype='float')
abs_trains_sizes,train_scores, test_scores = learning_curve(LinearSVC(),
        X, y,cv=10, scoring="accuracy",train_sizes=train_sizes)
###### 对每个 C ，获取 10 折交叉上的预测得分上的均值和方差 #####
train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)
####### 绘图 ######
fig=plt.figure()
ax=fig.add_subplot(1,1,1)

ax.plot(abs_trains_sizes, train_scores_mean, label="Training Accuracy", color="r")
ax.fill_between(abs_trains_sizes, train_scores_mean - train_scores_std,
                 train_scores_mean + train_scores_std, alpha=0.2, color="r")
ax.plot(abs_trains_sizes, test_scores_mean, label="Testing Accuracy", color="g")
ax.fill_between(abs_trains_sizes, test_scores_mean - test_scores_std,
                 test_scores_mean + test_scores_std, alpha=0.2, color="g")

ax.set_title("Learning Curve with LinearSVC")
ax.set_xlabel("Sample Nums")
ax.set_ylabel("Score")
ax.set_ylim(0,1.1)
ax.legend(loc='best')
plt.show()
```

这里使用的数据集是scikit-learn自带的digits数据集。使用线性支持向量机作为分类器。使用准确率作为评分标准，且使用 10 折交叉进行多次训练。

由于采用了 10 折交叉，因此对于每一个大小的数据集，其训练得分和预测得分均为 10 个。使用了这 10 个得分的均值和标准差，运行结果如图 12.7 所示。

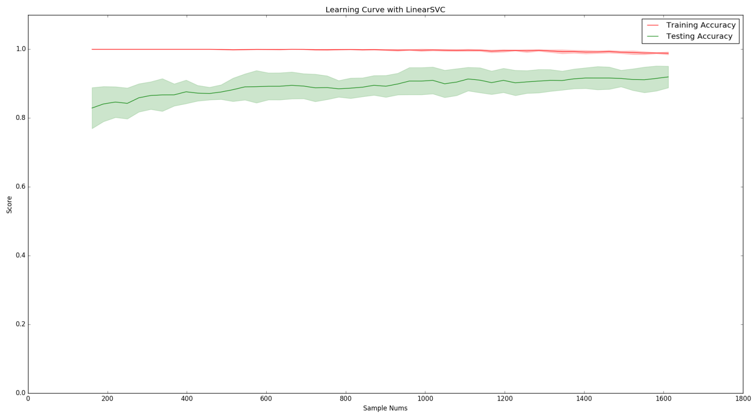


图 12.7 learning_curve

12.3.4 参数优化

自动化调参进行参数优化是使用 sklearn 优雅地进行机器学习的核心，自动化调参技术帮我们省去了人工调参的烦琐和经验不足。

暴力搜索寻优 GridSearchCV

网格搜索为自动化调参的常见技术之一，grid_search 包提供了自动化调参的工具，包括 GridSearchCV 类。GridSearchCV 根据给定的模型自动进行交叉验证，通过调节每一个参数来跟踪评分结果，实际上，该过程代替了进行参数搜索时的 for 循环过程。

scikit-learn提供了GridSearchCV来实现参数优化。其原型为

```
class sklearn.model_selection.GridSearchCV(estimator, param_grid, scoring=None,
    fit_params=None, n_jobs=1, iid=True, refit=True, cv=None,
      verbose=0, pre_dispatch='2*n_jobs', error_score='raise')
```

参数

- estimator：一个学习器对象。它必须有.fit方法用于学习，.predict方法用于预测，有.score方法用于性能评分。
- param_grid：字典或者字典的列表。每个字典都给出了学习器的一个参数，其中
  - 字典的键就是参数名。
  - 字典的值是一个列表，指定了参数对应的候选值序列。
- scoring：一个字符串，或者可调用对象，或者None。它指定了评分函数，其原型是：scorer(estimator, X, y)。如果为None，则使用学习器的.score方法。如果为字符串，可以为下列字符串。

- 'accuracy'：采用的是metrics.accuracy_score评分函数。
- 'average_precision'：采用的是metrics.average_precision_score评分函数。
- f1系列：采用的是metrics.f1_score 评分函数，包括如下。
  - 'f1'：用于二类分类。
  - 'f1_micro'：micro-averaged。
  - 'f1_macro'：macro-averaged。
  - 'f1_weighted'：weighted average。
  - 'f1_samples'：by multilabel sample。
- 'log_loss'：采用的是metrics.log_loss评分函数。
- 'precision'系列，采用的是metrics.precision_score评分函数，具体形式类似f1系列。
- 'recall' 系列，采用的是metrics.recall_score评分函数，具体形式类似f1系列。
- 'roc_auc'：采用的是metrics.roc_auc_score评分函数。
- 'adjusted_rand_score'：采用的是metrics.adjusted_rand_score评分函数。
- 'mean_absolute_error'：采用的是metrics.mean_absolute_error评分函数。
- 'mean_squared_error’'：采用的是metrics.mean_squared_error评分函数。
- 'median_absolute_error'：采用的是metrics.median_absolute_error评分函数。
- 'r2'：采用的是metrics.r2_score评分函数。

- fit_params：一个字典，用来给学习器的.fit方法传递参数。
- iid：如果为True，则表示数据是独立同分布的。
- cv：一个整数、k 折交叉生成器、一个迭代器、或者None。
  - 如果为None，则使用默认的 3 折交叉生成器。
  - 如果为整数，则指定了 k 折交叉生成器的 k 值。
  - 如果为 k 折交叉生成器，则直接指定了 k 折交叉生成器。
  - 如果为迭代器，则迭代器的结果就是数据集划分的结果。
- refit：一个布尔值，如果为True，则在参数优化之后使用整个数据集来重新训练该最优的estimator。
- verbose：一个整数。它控制输出日志的内容。该值越大，输出越多的内容。
- n_jobs：并行性。默认为 −1 表示派发任务到所有计算机的 CPU 上。
- pre_dispatch：一个整数或者字符串，用于控制并行执行时分发的总的任务数量。
- error_score：一个数值或者字符串'raise'，指定当estimator训练发生异常时，如何处理。
  - 如果为'raise'，则抛出异常。
  - 如果为数值，则将该数值作为本轮estimator的预测得分。

属性

- grid_scores_：命名元组组成的列表，列表中每个元素都对应了一个参数组合的测试得分。
- best_estimator_：一个学习器对象，代表根据候选参数组合筛选出来的最佳的学习器。

- best_score_：最佳学习器的性能评分。
- best_params_：最佳参数组合。

方法

- fit(X[, y])：执行参数优化。
- predict(X)：使用学到的最佳学习器来预测数据。
- predict_log_proba(X)：使用学到的最佳学习器来预测数据为各类别的概率的对数值。
- predict_proba(X)：使用学到的最佳学习器来预测数据为各类别的概率。
- score(X[, y])：通过给定的数据集来判断学到的最佳学习器的预测性能。

GridSearchCV实现了estimator的.fit、.score方法。这些方法内部会调用estimator的对应的方法。在调用.fit方法时，首先会将训练集进行 k 折交叉，然后在每次划分的集合上进行多轮的训练和验证（每一轮都采用一种参数组合）。

示例：

```
from sklearn.datasets import load_digits
from sklearn.linear_model import  LogisticRegression
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split
### 加载数据
digits = load_digits()
X_train,X_test,y_train,y_test=train_test_split(digits.data, digits.target,test_size=0.25,
            random_state=0,stratify=digits.target)
#### 参数优化 ######
tuned_parameters = [{'penalty': ['l1','l2'],
                    'C': [0.01,0.05,0.1,0.5,1,5,10,50,100],
                    'solver':['liblinear'],
                    'multi_class': ['ovr']},

                    {'penalty': ['l2'],
                    'C': [0.01,0.05,0.1,0.5,1,5,10,50,100],
                     'solver':['lbfgs'],
                    'multi_class': ['ovr','multinomial']},
                    ]
clf=GridSearchCV(LogisticRegression(tol=1e-6),tuned_parameters,cv=10)
clf.fit(X_train,y_train)
print("Best parameters set found:",clf.best_params_)
print("Grid scores:")
for params, mean_score, scores in clf.grid_scores_:
         print("\t%0.3f (+/-%0.03f) for %s" % (mean_score, scores.std() * 2, params))

print("Optimized Score:",clf.score(X_test,y_test))
print("Detailed classification report:")
```

```
y_true, y_pred = y_test, clf.predict(X_test)
print(classification_report(y_true, y_pred))
```

这里使用的数据集是scikit-learn自带的digits数据集。使用对数概率回归模型作为分类器，要优化的参数如下。

- penalty：正则化形式，可以为 'l1'（表示 L_1 正则化）、'l2'（表示 L_2 正则化）。
- C：正则化系数。理论上其取值范围是 $0 \to \infty$，这里取 0.01,0.05,0.1,0.5,1,5,10,50,100 这几个有限的值。
- solver：求解器的形式，这里选取 liblinear,lbfgs。
- multi_class：多分类问题的求解策略，可以为 ovr,multinomial。

因为某些参数不能共存，如 penalty='l1'不能与solver='lbfgs'共存（拟牛顿算法要求使用 L_2 正则化形式）；multi_class='multinomial' 不能与 solver='liblinear'共存。因此提供了两个字典作为一个列表，传给GridSearchCV作为待学习的参数。

在GridSearchCV中使用准确率作为评价标准，使用 10 折交叉生成器，将原始数据分层采样法划分成训练集与测试集，将训练集传给GridSearchCV用于参数优化。

由于使用了 10 折交叉生成器，每一种参数的组合将会迭代 10 次（每次使用其中的一折作为测试集，其他作为训练集）。因此我们对每一种参数的组合，打印出了这 10 次的测试得分的均值和标准差。

运行结果如下所示：

```
Best parameters set found: {'multi_class': 'multinomial', 'penalty': 'l2',
        'C': 0.05, 'solver': 'lbfgs'}
Grid scores:
    0.930 (+/-0.041) for {'multi_class': 'ovr', 'penalty': 'l1', 'C': 0.01,
            'solver': 'liblinear'}
    0.965 (+/-0.019) for {'multi_class': 'ovr', 'penalty': 'l2', 'C': 0.01,
            'solver': 'liblinear'}
    0.965 (+/-0.023) for {'multi_class': 'ovr', 'penalty': 'l1', 'C': 0.05,
            'solver': 'liblinear'}
    0.966 (+/-0.025) for {'multi_class': 'ovr', 'penalty': 'l2', 'C': 0.05,
             'solver': 'liblinear'}
    0.965 (+/-0.023) for {'multi_class': 'ovr', 'penalty': 'l1', 'C': 0.1,
            'solver': 'liblinear'}
    0.963 (+/-0.029) for {'multi_class': 'ovr', 'penalty': 'l2', 'C': 0.1,
             'solver': 'liblinear'}
    0.955 (+/-0.031) for {'multi_class': 'ovr', 'penalty': 'l1', 'C': 0.5,
            'solver': 'liblinear'}
    0.958 (+/-0.029) for {'multi_class': 'ovr', 'penalty': 'l2', 'C': 0.5,
            'solver': 'liblinear'}
    0.955 (+/-0.038) for {'multi_class': 'ovr', 'penalty': 'l1', 'C': 1,
            'solver': 'liblinear'}
```

```
0.957 (+/-0.036) for {'multi_class': 'ovr', 'penalty': 'l2', 'C': 1,
        'solver': 'liblinear'}
0.952 (+/-0.038) for {'multi_class': 'ovr', 'penalty': 'l1', 'C': 5,
        'solver': 'liblinear'}
0.955 (+/-0.043) for {'multi_class': 'ovr', 'penalty': 'l2', 'C': 5,
        'solver': 'liblinear'}
0.949 (+/-0.043) for {'multi_class': 'ovr', 'penalty': 'l1', 'C': 10,
         'solver': 'liblinear'}
0.951 (+/-0.052) for {'multi_class': 'ovr', 'penalty': 'l2', 'C': 10,
        'solver': 'liblinear'}
0.947 (+/-0.042) for {'multi_class': 'ovr', 'penalty': 'l1', 'C': 50,
        'solver': 'liblinear'}
0.941 (+/-0.052) for {'multi_class': 'ovr', 'penalty': 'l2', 'C': 50,
        'solver': 'liblinear'}
0.948 (+/-0.040) for {'multi_class': 'ovr', 'penalty': 'l1', 'C': 100,
        'solver': 'liblinear'}
0.940 (+/-0.054) for {'multi_class': 'ovr', 'penalty': 'l2', 'C': 100,
        'solver': 'liblinear'}
0.966 (+/-0.021) for {'multi_class': 'ovr', 'penalty': 'l2', 'C': 0.01,
        'solver': 'lbfgs'}
0.967 (+/-0.023) for {'multi_class': 'multinomial', 'penalty': 'l2',
         'C': 0.01, 'solver': 'lbfgs'}
0.967 (+/-0.023) for {'multi_class': 'ovr', 'penalty': 'l2', 'C': 0.05,
        'solver': 'lbfgs'}
0.973 (+/-0.027) for {'multi_class': 'multinomial', 'penalty': 'l2',
        'C': 0.05, 'solver': 'lbfgs'}
0.964 (+/-0.021) for {'multi_class': 'ovr', 'penalty': 'l2', 'C': 0.1,
        'solver': 'lbfgs'}
0.970 (+/-0.029) for {'multi_class': 'multinomial', 'penalty': 'l2',
        'C': 0.1, 'solver': 'lbfgs'}
0.955 (+/-0.030) for {'multi_class': 'ovr', 'penalty': 'l2', 'C': 0.5,
        'solver': 'lbfgs'}
0.965 (+/-0.033) for {'multi_class': 'multinomial', 'penalty': 'l2',
         'C': 0.5, 'solver': 'lbfgs'}
0.956 (+/-0.035) for {'multi_class': 'ovr', 'penalty': 'l2', 'C': 1,
        'solver': 'lbfgs'}
0.965 (+/-0.029) for {'multi_class': 'multinomial', 'penalty': 'l2',
        'C': 1, 'solver': 'lbfgs'}
0.950 (+/-0.048) for {'multi_class': 'ovr', 'penalty': 'l2', 'C': 5,
         'solver': 'lbfgs'}
0.965 (+/-0.033) for {'multi_class': 'multinomial', 'penalty': 'l2',
        'C': 5, 'solver': 'lbfgs'}
0.949 (+/-0.046) for {'multi_class': 'ovr', 'penalty': 'l2', 'C': 10,
        'solver': 'lbfgs'}
0.963 (+/-0.030) for {'multi_class': 'multinomial', 'penalty': 'l2',
```

```
          'C': 10, 'solver': 'lbfgs'}
    0.941 (+/-0.053) for {'multi_class': 'ovr', 'penalty': 'l2', 'C': 50,
           'solver': 'lbfgs'}
    0.961 (+/-0.028) for {'multi_class': 'multinomial', 'penalty': 'l2',
          'C': 50, 'solver': 'lbfgs'}
    0.938 (+/-0.051) for {'multi_class': 'ovr', 'penalty': 'l2', 'C': 100,
          'solver': 'lbfgs'}
    0.961 (+/-0.030) for {'multi_class': 'multinomial', 'penalty': 'l2',
          'C': 100, 'solver': 'lbfgs'}
Optimized Score: 0.966740576497
Detailed classification report:
             precision    recall  f1-score   support

          0       1.00      0.98      0.99        45
          1       0.85      0.98      0.91        46
          2       1.00      0.98      0.99        44
          3       0.98      1.00      0.99        46
          4       0.98      0.93      0.95        45
          5       0.98      0.93      0.96        46
          6       1.00      0.98      0.99        45
          7       0.98      0.98      0.98        45
          8       0.95      0.93      0.94        44
          9       0.98      0.98      0.98        45

avg / total       0.97      0.97      0.97       451
```

可以看到，最佳的参数为：C= 0.05, penalty='l2', solver= 'lbfgs', multi_class='multinomial'，此时的学习器对预测集的测试准确率为 96.6740576497 %。还给出了最佳参数情况下对数概率回归模型的Classification Report。

> 从结果中发现Grid scores中有一行：
>
> ```
> 0.973 (+/-0.027) for {'multi_class': 'multinomial', 'penalty': 'l2',
>        'C': 0.05, 'solver': 'lbfgs'}
> ```
>
> 这是因为将原始数据集拆分成训练集和测试集，GridSearchCV使用的是训练集，而 96.6740576497% 是在测试集上获得的预测准确率，两者使用的是不同的数据。

随机搜索寻优（RandomizedSearchCV）

GridSearchCV采用的是暴力寻找的方法来寻找最优参数。当待优化的参数是离散的取值的时候，GridSearchCV能够顺利地找出最优的参数。但是当待优化的参数是连续取值的时候，暴力寻找就有心无力了。GridSearchCV的做法是从这些连续值中挑选几个值作为代表，从而

在这些代表中挑选出最佳的参数。

scikit-learn提供的RandomizedSearchCV采用随机搜索所有的候选参数对的方法来寻找最优的参数组合。它是另一种参数寻找方式。其原型为：

```
class sklearn.model_selection.RandomizedSearchCV(estimator, param_distributions,
 n_iter=10, scoring=None, fit_params=None, n_jobs=1, iid=True, refit=True,
 cv=None, verbose=0, pre_dispatch='2*n_jobs', random_state=None, error_score='raise')
```

参数

- estimator：一个学习器对象。它必须有.fit方法用于学习，.predict方法用于预测，有.score方法用于性能评分。
- param_distributions：字典或者字典的列表。每个字典都给出了学习器的一个参数，其中
  - 字典的键就是参数名。
  - 字典的值是一个分布类，分布类必须提供.rvs方法。通常可以使用scipy.stats模块中提供的分布类，比如scipy.expon(指数分布)、scipy.gamma(gamma 分布)、scipy.uniform(均匀分布)、randint，等等。
  - 字典的值也可以是一个数值序列。此时就在该序列中均匀采样。
- n_iter：一个整数，指定每个参数采样的数量。通常该值越大，参数优化的效果越好。但是参数越大，运行时间也更长。
- scoring：一个字符串，或者可调用对象，或者None。它指定了评分函数，其原型是：scorer(estimator, X, y)。如果为None，则使用学习器的.score方法。如果为字符串，可以为下列字符串。
  - 'accuracy': 采用的是metrics.accuracy_score评分函数。
  - 'average_precision'：采用的是metrics.average_precision_score评分函数。
  - f1系列：采用的是metrics.f1_score 评分函数，包括
    - 'f1'：用于二类分类。
    - 'f1_micro' ：micro-averaged。
    - 'f1_macro'：macro-averaged。
    - 'f1_weighted'：weighted average。
    - 'f1_samples' ：by multilabel sample。
  - 'log_loss'：采用的是metrics.log_loss评分函数。
  - 'precision'系列，采用的是metrics.precision_score评分函数，具体形式类似f1系列。
  - 'recall' 系列，采用的是metrics.recall_score评分函数，具体形式类似f1系列。
  - 'roc_auc'：采用的是metrics.roc_auc_score 评分函数。
  - 'adjusted_rand_score'：采用的是metrics.adjusted_rand_score 评分函数。
  - 'mean_absolute_error'：采用的是metrics.mean_absolute_error 评分函数。
  - 'mean_squared_error'：采用的是metrics.mean_squared_error 评分函数。
  - 'median_absolute_error'：采用的是metrics.median_absolute_error 评分函数。

  - 'r2'：采用的是metrics.r2_score 评分函数。
- fit_params：一个字典，用来给学习器的.fit方法传递参数。
- iid：如果为True，则表示数据是独立同分布的。
- cv：一个整数、k 折交叉生成器、一个迭代器、或者None。
  - 如果为None，则使用默认的 3 折交叉生成器。
  - 如果为整数，则指定了 k 折交叉生成器的 k 值。
  - 如果为 k 折交叉生成器，则直接指定了 k 折交叉生成器。
  - 如果为迭代器，则迭代器的结果就是数据集划分的结果。
- refit：一个布尔值，如果为True，则在参数优化之后使用整个数据集来重新训练该最优的estimator。
- verbose：一个整数。它控制输出日志的内容。该值越大，输出越多的内容。
- n_jobs：并行性。默认为 −1 表示派发任务到所有计算机的 CPU 上。
- pre_dispatch：一个整数或者字符串，用于控制并行执行时分发的总的任务数量。
- random_state：一个整数或者一个RandomState实例，或者None，用于待优化参数的采样过程。
  - 如果为整数，则它指定了随机数生成器的种子。
  - 如果为RandomState实例，则指定了随机数生成器。
  - 如果为None，则使用默认的随机数生成器。
- error_score：一个数值或者字符串'raise'，指定当estimator训练发生异常时如何处理。
  - 如果为'raise'，则抛出异常。
  - 如果为数值，则将该数值作为本轮estimator的预测得分。

属性

- grid_scores_：命名元组组成的列表，列表中每个元素都对应了一个参数组合的测试得分。
- best_estimator_：一个学习器对象，代表了根据候选参数组合筛选出来的最佳的学习器。
- best_score_：最佳学习器的性能评分。
- best_params_：最佳参数组合。

方法

- fit(X[, y])：执行参数优化。
- predict(X)：使用学到的最佳学习器来预测数据。
- predict_log_proba(X)：使用学到的最佳学习器来预测数据为各类别的概率的对数值。
- predict_proba(X)：使用学到的最佳学习器来预测数据为各类别的概率。
- score(X[, y])：通过给定的数据集来判断学到的最佳学习器的预测性能。

RandomizedSearchCV的接口和用法类似 GridSearchCV。

示例：

```
from sklearn.datasets import load_digits
from sklearn.linear_model import  LogisticRegression
from sklearn.model_selection import RandomizedSearchCV
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split
import  scipy
### 加载数据
digits = load_digits()
X_train,X_test,y_train,y_test=train_test_split(digits.data, digits.target,
            test_size=0.25,random_state=0,stratify=digits.target)
#### 参数优化 ######
tuned_parameters ={  'C': scipy.stats.expon(scale=100),
                    'multi_class': ['ovr','multinomial']}
clf=RandomizedSearchCV(LogisticRegression(penalty='l2',solver='lbfgs',tol=1e-6),
                    tuned_parameters,cv=10,scoring="accuracy",n_iter=100)
clf.fit(X_train,y_train)
print("Best parameters set found:",clf.best_params_)
print("Randomized Grid scores:")
for params, mean_score, scores in clf.grid_scores_:
         print("\t%0.3f (+/-%0.03f) for %s" % (mean_score, scores.std() * 2, params))

print("Optimized Score:",clf.score(X_test,y_test))
print("Detailed classification report:")
y_true, y_pred = y_test, clf.predict(X_test)
print(classification_report(y_true, y_pred))
```

这里使用的数据集是 scikit-learn 自带的digits‘数据集。使用对数概率回归模型作为分类器。要优化的参数如下。

- ❑ C：正则化系数。理论上其取值范围是 $0 \to \infty$，这里取其分布为指数分布，如图 12.8 所示。

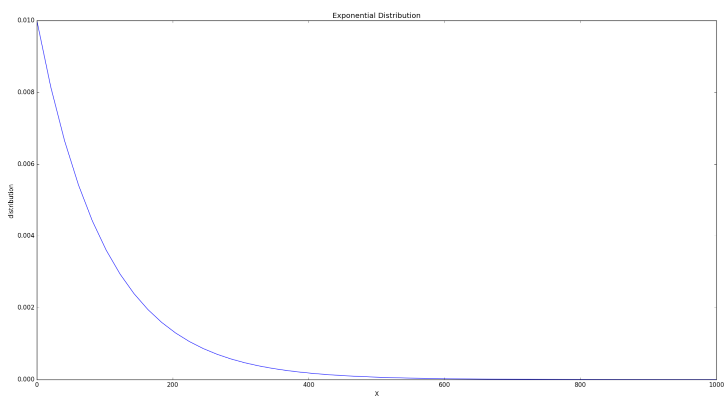


图 12.8 exponential_distribution

- multi_class：多分类问题的求解策略，可以为 ovr,multinomial。

在RandomizedSearchCV中使用准确率作为评价标准，使用 10 折交叉生成器。将原始数据分层采样法划分成训练集与测试集。将训练集传给RandomizedSearchCV用于参数优化。

由于使用了 10 折交叉生成器，每一种参数的组合将会迭代 10 次（每次使用其中的一折作为测试集，其他作为训练集）。因此我们对每一种参数的组合，打印出了这 10 次测试得分的均值和标准差。

运行结果如下所示：

```
Best parameters set found: {'C': 0.15760523636562604, 'multi_class': 'multinomial'}
Randomized Grid scores:
    0.960 (+/-0.029) for {'C': 419.6106828594601, 'multi_class': 'multinomial'}
    0.935 (+/-0.052) for {'C': 159.85496896412525, 'multi_class': 'ovr'}
    0.935 (+/-0.047) for {'C': 312.91681440547967, 'multi_class': 'ovr'}
    0.943 (+/-0.051) for {'C': 37.73490142022267, 'multi_class': 'ovr'}
    0.961 (+/-0.030) for {'C': 66.95998927301457, 'multi_class': 'multinomial'}
    0.944 (+/-0.050) for {'C': 29.82991465224613, 'multi_class': 'ovr'}
    0.961 (+/-0.029) for {'C': 270.8914633037311, 'multi_class': 'multinomial'}
    0.961 (+/-0.028) for {'C': 105.14412204779137, 'multi_class': 'multinomial'}
    0.941 (+/-0.055) for {'C': 93.12803396355399, 'multi_class': 'ovr'}
    0.960 (+/-0.033) for {'C': 97.44201351191376, 'multi_class': 'multinomial'}
    0.936 (+/-0.045) for {'C': 376.8797005297582, 'multi_class': 'ovr'}
    0.941 (+/-0.053) for {'C': 35.916965629723016, 'multi_class': 'ovr'}
    0.958 (+/-0.031) for {'C': 147.99951954938956, 'multi_class': 'multinomial'}
    0.961 (+/-0.030) for {'C': 58.01558576309599, 'multi_class': 'multinomial'}
    0.967 (+/-0.032) for {'C': 0.7859129455762149, 'multi_class': 'multinomial'}
    0.940 (+/-0.053) for {'C': 79.90696182021426, 'multi_class': 'ovr'}
    0.937 (+/-0.051) for {'C': 163.60549607172769, 'multi_class': 'ovr'}
    0.936 (+/-0.050) for {'C': 124.30829674906263, 'multi_class': 'ovr'}
    0.947 (+/-0.046) for {'C': 14.79922527246503, 'multi_class': 'ovr'}
    0.961 (+/-0.027) for {'C': 28.515523927635368, 'multi_class': 'multinomial'}
    0.944 (+/-0.053) for {'C': 25.419852438851827, 'multi_class': 'ovr'}
    0.961 (+/-0.030) for {'C': 29.488987477371687, 'multi_class': 'multinomial'}
    0.958 (+/-0.034) for {'C': 275.032823148553, 'multi_class': 'multinomial'}
    0.940 (+/-0.058) for {'C': 42.38863241082116, 'multi_class': 'ovr'}
    0.961 (+/-0.030) for {'C': 67.13926893789632, 'multi_class': 'multinomial'}
    0.942 (+/-0.051) for {'C': 45.02950640329191, 'multi_class': 'ovr'}
    0.962 (+/-0.028) for {'C': 32.337309333426425, 'multi_class': 'multinomial'}
    0.940 (+/-0.052) for {'C': 81.014802830513, 'multi_class': 'ovr'}
    0.942 (+/-0.055) for {'C': 34.7911916681384, 'multi_class': 'ovr'}
    0.941 (+/-0.050) for {'C': 55.05196767376429, 'multi_class': 'ovr'}
    0.964 (+/-0.032) for {'C': 2.9522052075751337, 'multi_class': 'multinomial'}
    0.946 (+/-0.048) for {'C': 12.847571013059959, 'multi_class': 'ovr'}
    0.933 (+/-0.048) for {'C': 189.1840995097719, 'multi_class': 'ovr'}
    0.939 (+/-0.052) for {'C': 89.48416510042183, 'multi_class': 'ovr'}
```

```
0.965 (+/-0.035) for {'C': 1.2253248733083697, 'multi_class': 'multinomial'}
0.935 (+/-0.048) for {'C': 156.3139150032594, 'multi_class': 'ovr'}
0.962 (+/-0.028) for {'C': 33.294639706455484, 'multi_class': 'multinomial'}
0.936 (+/-0.049) for {'C': 170.00145351334257, 'multi_class': 'ovr'}
0.961 (+/-0.033) for {'C': 98.08200150941379, 'multi_class': 'multinomial'}
0.960 (+/-0.033) for {'C': 46.77801460706446, 'multi_class': 'multinomial'}
0.961 (+/-0.032) for {'C': 299.46494588415885, 'multi_class': 'multinomial'}
0.938 (+/-0.049) for {'C': 60.54163434906271, 'multi_class': 'ovr'}
0.941 (+/-0.050) for {'C': 62.36156304059989, 'multi_class': 'ovr'}
0.938 (+/-0.053) for {'C': 153.88175760600794, 'multi_class': 'ovr'}
0.949 (+/-0.047) for {'C': 9.918077225260271, 'multi_class': 'ovr'}
0.959 (+/-0.032) for {'C': 259.0222201682972, 'multi_class': 'multinomial'}
0.939 (+/-0.050) for {'C': 59.86361320656244, 'multi_class': 'ovr'}
0.958 (+/-0.034) for {'C': 62.77669486908424, 'multi_class': 'multinomial'}
0.961 (+/-0.029) for {'C': 96.61671069556816, 'multi_class': 'multinomial'}
0.937 (+/-0.051) for {'C': 134.40001702048565, 'multi_class': 'ovr'}
0.935 (+/-0.044) for {'C': 494.30937438185305, 'multi_class': 'ovr'}
0.964 (+/-0.032) for {'C': 12.840678932466892, 'multi_class': 'multinomial'}
0.939 (+/-0.053) for {'C': 61.130948637456854, 'multi_class': 'ovr'}
0.939 (+/-0.049) for {'C': 97.60837744671288, 'multi_class': 'ovr'}
0.943 (+/-0.049) for {'C': 44.51488961942965, 'multi_class': 'ovr'}
0.960 (+/-0.029) for {'C': 99.20816203434867, 'multi_class': 'multinomial'}
0.962 (+/-0.030) for {'C': 9.78450292598448, 'multi_class': 'multinomial'}
0.942 (+/-0.052) for {'C': 27.377776570880158, 'multi_class': 'ovr'}
0.936 (+/-0.048) for {'C': 223.77578893600503, 'multi_class': 'ovr'}
0.958 (+/-0.034) for {'C': 228.74477404697657, 'multi_class': 'multinomial'}
0.959 (+/-0.034) for {'C': 238.54470736852176, 'multi_class': 'multinomial'}
0.962 (+/-0.030) for {'C': 38.28353149269131, 'multi_class': 'multinomial'}
0.941 (+/-0.056) for {'C': 44.59514144495703, 'multi_class': 'ovr'}
0.964 (+/-0.032) for {'C': 9.014208234776055, 'multi_class': 'multinomial'}
0.942 (+/-0.053) for {'C': 38.37991749969876, 'multi_class': 'ovr'}
0.938 (+/-0.057) for {'C': 93.52061177559182, 'multi_class': 'ovr'}
0.961 (+/-0.032) for {'C': 425.49837360405877, 'multi_class': 'multinomial'}
0.936 (+/-0.051) for {'C': 144.11794501875585, 'multi_class': 'ovr'}
0.961 (+/-0.028) for {'C': 34.53877740110517, 'multi_class': 'multinomial'}
0.964 (+/-0.030) for {'C': 12.697763547765572, 'multi_class': 'multinomial'}
0.960 (+/-0.029) for {'C': 168.17579080743704, 'multi_class': 'multinomial'}
0.948 (+/-0.044) for {'C': 15.779059135468609, 'multi_class': 'ovr'}
0.960 (+/-0.030) for {'C': 74.10800865885146, 'multi_class': 'multinomial'}
0.950 (+/-0.045) for {'C': 4.250550647386843, 'multi_class': 'ovr'}
0.945 (+/-0.048) for {'C': 17.323726922772135, 'multi_class': 'ovr'}
0.941 (+/-0.054) for {'C': 35.285452491876214, 'multi_class': 'ovr'}
0.954 (+/-0.039) for {'C': 3.1088320193006034, 'multi_class': 'ovr'}
0.961 (+/-0.028) for {'C': 58.78813867534984, 'multi_class': 'multinomial'}
0.940 (+/-0.055) for {'C': 36.1845944791761, 'multi_class': 'ovr'}
```

```
    0.942 (+/-0.057) for {'C': 29.962656984838414, 'multi_class': 'ovr'}
    0.958 (+/-0.030) for {'C': 153.435683523197, 'multi_class': 'multinomial'}
    0.941 (+/-0.054) for {'C': 38.79383407714137, 'multi_class': 'ovr'}
    0.962 (+/-0.027) for {'C': 21.985418652801386, 'multi_class': 'multinomial'}
    0.941 (+/-0.056) for {'C': 31.29754481810053, 'multi_class': 'ovr'}
    0.944 (+/-0.051) for {'C': 23.018536704466175, 'multi_class': 'ovr'}
    0.936 (+/-0.050) for {'C': 117.77339925819594, 'multi_class': 'ovr'}
    0.952 (+/-0.044) for {'C': 3.8840686833693816, 'multi_class': 'ovr'}
    0.935 (+/-0.047) for {'C': 182.15121974096243, 'multi_class': 'ovr'}
    0.961 (+/-0.029) for {'C': 117.03346175893665, 'multi_class': 'multinomial'}
    0.960 (+/-0.028) for {'C': 144.1506849587325, 'multi_class': 'multinomial'}
    0.939 (+/-0.058) for {'C': 34.56667908593655, 'multi_class': 'ovr'}
    0.970 (+/-0.028) for {'C': 0.15760523636562604, 'multi_class': 'multinomial'}
    0.952 (+/-0.046) for {'C': 3.974439416587531, 'multi_class': 'ovr'}
    0.961 (+/-0.033) for {'C': 357.3003606793094, 'multi_class': 'multinomial'}
    0.959 (+/-0.031) for {'C': 210.58831003589216, 'multi_class': 'multinomial'}
    0.935 (+/-0.045) for {'C': 178.26428793294028, 'multi_class': 'ovr'}
    0.960 (+/-0.038) for {'C': 579.2275101348413, 'multi_class': 'multinomial'}
    0.962 (+/-0.028) for {'C': 31.7787726303104, 'multi_class': 'multinomial'}
    0.960 (+/-0.033) for {'C': 56.74854052868679, 'multi_class': 'multinomial'}
    0.961 (+/-0.028) for {'C': 161.23245181640476, 'multi_class': 'multinomial'}
Optimized Score: 0.962305986696
Detailed classification report:
             precision    recall  f1-score   support

          0       1.00      0.98      0.99        45
          1       0.87      0.98      0.92        46
          2       1.00      0.98      0.99        44
          3       0.94      1.00      0.97        46
          4       0.98      0.96      0.97        45
          5       0.98      0.93      0.96        46
          6       1.00      0.98      0.99        45
          7       0.98      0.96      0.97        45
          8       0.95      0.91      0.93        44
          9       0.96      0.96      0.96        45

avg / total       0.96      0.96      0.96       451
```

可以看到，最佳的参数为：C=0.08445721387071305, multi_class='multinomial'，此时的学习器对预测集的测试准确率为 96.2305986696 %。还给出了最佳参数情况下的对数概率回归模型的Classification Report。

使用RandomizedSearchCV找到的最优参数，使用该参数的学习器在测试集上的预测准确率为 96.2305986696%。使用GridSearchCV 找到的最优参数，使用该参数的学习器在测试集上的预测准确率为 96.6740576497%。此时 RandomizedSearchCV的效果不如 GridSearchCV。但是

RandomizedSearchCV采用随机采样,有无数的参数组合,更有可能找到最佳的参数组合(GridSearchCV 只能选择有限的参数组合)。

第四篇

Kaggle 实战篇

第 13 章

Kaggle 牛刀小试

13.1 Kaggle 简介

Kaggle 是一个数据分析的竞赛平台（网址为 https://www.kaggle.com），于 2010 年成立，短短几年，便风靡全球的数据科学圈。企业或者研究者可以将数据、问题描述、期望的指标发布到Kaggle上，以竞赛的形式向广大的数据科学家征集解决方案。Kaggle上的参赛者将数据下载下来，分析数据，然后运用机器学习的知识，建立算法模型，得出结果。参赛者将结果提交之后，如果提交的结果符合指标要求并且在参赛者中排名第一，则将获得丰厚的奖金。

Kaggle 对于喜欢边学边做（而不是通过读书或者看讲座）的人来说是一个非常好的入门方式，本章就以Kaggle上的一个竞赛为例阐述如何遴选最佳的算法模型。该竞赛的数据由Red Hat公司提供。Red Hat收集了大量的用户数据，该公司希望创建一个这样的分类算法：能准确识别客户的特点从而筛选出有商业价值的客户。竞赛的地址为https://www.Kaggle.com/c/predicting-red-hat-business-value，有效期为 2016 年 8 月 1 日到 2016 年 9 月 19 日。

Red Hat提供的数据下载地址为https://www.Kaggle.com/c/predicting-red-hat-business-value/data。数据使用两个独立的文件，需要将它们合并成一个文件。这两个文件介绍如下。

- people.csv：给出了用户的特征。每一行代表一个用户，其中每个用户都有一个唯一的people_id。
- act_train.csv：给出了行为activity的特征以及标签label。每一行代表一个行为，其中每个行为都有一个唯一的activity_id。可以通过people_id这个键，将两个文件联合起来。

需要预测的是：一个特定的用户在某种特定行为下是否具有商业价值。在act_train.csv文件中，最后一行给出了监督信息：yes代表有商业价值，no代表没有商业价值。另外，Red Hat提示你：有几种不同类型的activity。其中：Type 1类型的activity不同于type 2-7的activity，因

为Type 1类型的activity拥有更多特征。

注意：根据官方的说明，在数据集中，people.csv的char_38列是连续数值类型，其他的数据都是离散集合类型。

最后还有两个额外的文件。

- act_test.csv：它给出了测试集的数据。你学到的模型需要对该文件代表的样本集进行预测，预测结果提交到Kaggle上。它与act_train.csv的唯一区别为：它没有标记信息。
- sample_submission.csv：它给出了一个模板，你提交的结果文件的格式要跟该文件的格式相同。

13.2 清洗数据

首先我们建立一个项目Kaggle,然后将下载的几个文件解压缩,放置到新建的文件夹Kaggle/data下。目录结构为：

```
Kaggle/
    data/
        act_test.csv
        act_train.csv
        people.csv
        sample_submission.csv
```

现在将开始进行对数据的探索过程。到当前为止，除了官方给出的数据说明外，我们对数据的特性一无所知。这里将使用jupyter notebook这个工具进行探索，它是一款非常优秀的展示与快速迭代工具。

> 但是在模型学习过程中，通常是把脚本直接提交到服务器中运行的。

13.2.1 加载数据

将使用pandas这个著名的第三方包来清洗数据，首先观察文件格式。

- people.csv的前三行：

```
people_id,char_1,group_1,char_2,date,char_3,char_4,char_5,char_6,char_7,char_8,char_9,char_10
    ,char_11,char_12,char_13,char_14,char_15,char_16,char_17,char_18,char_19,char_20,char_21
    ,char_22,char_23,char_24,char_25,char_26,char_27,char_28,char_29,char_30,char_31,char_32
    ,char_33,char_34,char_35,char_36,char_37,char_38
```

```
ppl_100,type 2,group 17304,type 2,2021-06-29,type 5,type 5,type 5,type 3,type 11,type 2,type
    2,True,False,False,True,True,False,True,False,False,False,False,True,False,False,False,
    False,False,True,True,False,True,True,False,False,True,True,True,False,36
ppl_100002,type 2,group 8688,type 3,2021-01-06,type 28,type 9,type 5,type 3,type 11,type 2,
    type 4,False,False,True,True,False,False,False,True,False,False,False,False,False,True,
    False,True,True,True,False,False,True,True,True,True,True,True,True,False,76
```

- act_train.csv的前三行：

```
people_id,activity_id,date,activity_category,char_1,char_2,char_3,char_4,char_5,char_6,char_7
    ,char_8,char_9,char_10,outcome
ppl_100,act2_1734928,2023-08-26,type 4,,,,,,,,,,type 76,0
ppl_100,act2_2434093,2022-09-27,type 2,,,,,,,,,,type 1,0
```

- act_test.csv的前三行：

```
people_id,activity_id,date,activity_category,char_1,char_2,char_3,char_4,char_5,char_6,char_7
    ,char_8,char_9,char_10
ppl_100004,act1_249281,2022-07-20,type 1,type 5,type 10,type 5,type 1,type 6,type 1,type 1,
    type 7,type 4,
ppl_100004,act2_230855,2022-07-20,type 5,,,,,,,,,,type 682
```

> sample_submission.csv与模型训练关系不大，因此暂时不处理它。

可以看到：分隔符为逗号','，第一行为标题行。调用pandas.read_csv()函数来加载数据。

- 加载people.csv:

```
people=pd.read_csv("data/people.csv",sep=',',header=0,keep_default_na=True,
    parse_dates=['date'])
people.set_index(keys=['people_id'],drop=True,append=False,inplace=True)
```

  header指定第一行为标题行。parse_dates给出了日期列，从而使得pandas直接将该列解析为numpy.datetime64类型。通过.set_index()方法将people_id列作为索引列。查看前 5 列数据：在jupyter notebook中运行people.head(5)，结果如图 13.1 所示。

- 加载act_train.csv:

```
act_train=pd.read_csv("data/act_train.csv",sep=',',header=0,keep_default_na=True,
    parse_dates=['date'])
act_train.set_index(keys=['people_id'],drop=True,append=False,inplace=True)
```

  header指定第一行为标题行。parse_dates给出了日期列，从而使得pandas直接将该列解析为numpy.datetime64类型。通过.set_index()方法将people_id列作为索引列。查看前 10 列数据：在jupyter notebook中运行act_train.head(10)，结果如图 13.2 所示。

- 加载 act_test.csv:

	char_1	group_1	char_2	date	char_3	char_4	char_5	char_6	char_7	char_8	...	char_29	char_30	char_31	char_32	cha
people_id																
ppl_100	type 2	group 17304	type 2	2021-06-29	type 5	type 5	type 5	type 3	type 11	type 2	...	False	True	True	False	Fals
ppl_100002	type 2	group 8688	type 3	2021-01-06	type 28	type 9	type 5	type 3	type 11	type 2	...	False	True	True	True	Tru
ppl_100003	type 2	group 33592	type 3	2022-06-10	type 4	type 8	type 5	type 2	type 5	type 2	...	False	False	True	True	Tru
ppl_100004	type 2	group 22593	type 3	2022-07-20	type 40	type 25	type 9	type 4	type 16	type 2	...	True	True	True	True	Tru
ppl_100006	type 2	group 6534	type 3	2022-07-27	type 40	type 25	type 9	type 3	type 8	type 2	...	False	False	True	False	Fals

5 rows × 40 columns

图 13.1 people_head

	activity_id	date	activity_category	char_1	char_2	char_3	char_4	char_5	char_6	char_7	char_8	char_9	char_10	out
people_id														
ppl_100	act2_1734928	2023-08-26	type 4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	type 76	0
ppl_100	act2_2434093	2022-09-27	type 2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	type 1	0
ppl_100	act2_3404049	2022-09-27	type 2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	type 1	0
ppl_100	act2_3651215	2023-08-04	type 2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	type 1	0
ppl_100	act2_4109017	2023-08-26	type 2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	type 1	0
ppl_100	act2_898576	2023-08-04	type 4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	type 1727	0
ppl_100002	act2_1233489	2022-11-23	type 2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	type 1	1
ppl_100002	act2_1623405	2022-11-23	type 2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	type 1	1
ppl_100003	act2_1111598	2023-02-07	type 2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	type 1	1
ppl_100003	act2_1177453	2023-06-28	type 2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	type 1	1

图 13.2 act_train_head

```
act_test=pd.read_csv("data/act_test.csv",sep=',',header=0,keep_default_na=True,
    parse_dates=['date'])
act_test.set_index(keys=['people_id'],drop=True,append=False,inplace=True)
```

header指定第一行为标题行。parse_dates给出了日期列，从而使得pandas直接将该列解析为numpy.datetime64类型。通过.set_index()方法将people_id列作为索引列。查看前 10 列数据：在jupyter notebook中运行act_test.head(10)，结果如图 13.3 所示。

	activity_id	date	activity_category	char_1	char_2	char_3	char_4	char_5	char_6	char_7	char_8	char_9	char_10
people_id													
ppl_100004	act1_249281	2022-07-20	type 1	type 5	type 10	type 5	type 1	type 6	type 1	type 1	type 7	type 4	NaN
ppl_100004	act2_230855	2022-07-20	type 5	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	type 682
ppl_10001	act1_240724	2022-10-14	type 1	type 12	type 1	type 5	type 4	type 6	type 1	type 1	type 13	type 10	NaN
ppl_10001	act1_83552	2022-11-27	type 1	type 20	type 10	type 5	type 4	type 6	type 1	type 1	type 5	type 5	NaN
ppl_10001	act2_1043301	2022-10-15	type 5	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	type 3015
ppl_10001	act2_112890	2022-11-27	type 5	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	type 4987
ppl_10001	act2_1169930	2022-10-15	type 5	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	type 3015
ppl_10001	act2_1924448	2022-10-15	type 5	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	type 3015
ppl_10001	act2_1953554	2022-10-15	type 5	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	type 3015
ppl_10001	act2_1971739	2022-11-28	type 5	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	type 3015

图 13.3 act_test_head

13.2.2 合并数据

通过DataFrame.merge()方法，以people_id为键，合并act数据和people数据。

- 合并act_train数据和people数据：

```
train_data=act_train.merge(people,how='left',left_index=True,right_index=True,
    suffixes=('_act', '_people'))
```

其中采用了左联合 (how='left')，基于行索引合并。对于相同的列标题，添加后缀'_act'和'_people'来加以区分。查看前 10 列数据：在jupyter notebook中运行train_data.head(10)，结果如图 13.4 所示。

	activity_id	date_act	activity_category	char_1_act	char_2_act	char_3_act	char_4_act	char_5_act	char_6_act	char_7_act
people_id										
ppl_100	act2_1734928	2023-08-26	type 4	NaN	NaN	NaN	NaN	NaN	NaN	NaN
ppl_100	act2_2434093	2022-09-27	type 2	NaN	NaN	NaN	NaN	NaN	NaN	NaN
ppl_100	act2_3404049	2022-09-27	type 2	NaN	NaN	NaN	NaN	NaN	NaN	NaN
ppl_100	act2_3651215	2023-08-04	type 2	NaN	NaN	NaN	NaN	NaN	NaN	NaN
ppl_100	act2_4109017	2023-08-26	type 2	NaN	NaN	NaN	NaN	NaN	NaN	NaN
ppl_100	act2_898576	2023-08-04	type 4	NaN	NaN	NaN	NaN	NaN	NaN	NaN
ppl_100002	act2_1233489	2022-11-23	type 2	NaN	NaN	NaN	NaN	NaN	NaN	NaN
ppl_100002	act2_1623405	2022-11-23	type 2	NaN	NaN	NaN	NaN	NaN	NaN	NaN
ppl_100003	act2_1111598	2023-02-07	type 2	NaN	NaN	NaN	NaN	NaN	NaN	NaN
ppl_100003	act2_1177453	2023-06-28	type 2	NaN	NaN	NaN	NaN	NaN	NaN	NaN

10 rows × 54 columns

图 13.4 train_data_head

- 合并act_test数据和people数据：

```
test_data=act_test.merge(people,how='left',left_index=True,right_index=True,
suffixes=('_act', '_people'))
```

各参数意义参考train_data。查看前 10 列数据如图 13.5 所示。

	activity_id	date_act	activity_category	char_1_act	char_2_act	char_3_act	char_4_act	char_5_act	char_6_act	char_7_act
people_id										
ppl_100004	act1_249281	2022-07-20	type 1	type 5	type 10	type 5	type 1	type 6	type 1	type 1
ppl_100004	act2_230855	2022-07-20	type 5	NaN	NaN	NaN	NaN	NaN	NaN	NaN
ppl_10001	act1_240724	2022-10-14	type 1	type 12	type 1	type 5	type 4	type 6	type 1	type 1
ppl_10001	act1_83552	2022-11-27	type 1	type 20	type 10	type 5	type 4	type 6	type 1	type 1
ppl_10001	act2_1043301	2022-10-15	type 5	NaN	NaN	NaN	NaN	NaN	NaN	NaN
ppl_10001	act2_112890	2022-11-27	type 5	NaN	NaN	NaN	NaN	NaN	NaN	NaN
ppl_10001	act2_1169930	2022-10-15	type 5	NaN	NaN	NaN	NaN	NaN	NaN	NaN
ppl_10001	act2_1924448	2022-10-15	type 5	NaN	NaN	NaN	NaN	NaN	NaN	NaN
ppl_10001	act2_1953554	2022-10-15	type 5	NaN	NaN	NaN	NaN	NaN	NaN	NaN
ppl_10001	act2_1971739	2022-11-28	type 5	NaN	NaN	NaN	NaN	NaN	NaN	NaN

10 rows × 53 columns

图 13.5 test_head

13.2.3 拆分数据

根据官方说明，type 1的activity和非type 1的activity的格式是不同的。根据这一提示，希望根据activity的type来进行拆分。

首先查看有哪些活动类型，以及分别由多少数据。在jupyter notebook中运行：train_data.activity_category.value_counts()，结果如下：

```
type 2    904683
type 5    490710
type 3    429408
type 4    207465
type 1    157615
type 6      4253
type 7      3157
Name: activity_category, dtype: int64
```

然后开始拆分数据，在jupyter notebook中执行代码：

```
types=['type %d'%i for i in range(1,8)]
train_datas={}
test_datas={}
for _type in types:
    train_datas[_type]=train_data[train_data.activity_category==_type]\
        .dropna(axis=(0,1), how='all')
    test_datas[_type]=test_data[test_data.activity_category==_type]\
        .dropna(axis=(0,1), how='all')
    print(train_datas[_type].activity_category.unique())
    print(test_datas[_type].activity_category.unique())
```

通过打印拆分后数据的activity_category列的唯一值来保证我们是正确拆分的。其中调用了DataFrame.dropna()方法，抛弃了那些全行为NaN的行，以及全列为NaN的列。

观察拆分后的数据，以train_datas中的type 1 ~3为例，如图 13.6 所示。

现在我们发现每一个数据集中，activity_category都不变。如train_datas['type 1']的 activity_category 列，全部都是type 1。因此将删除这种列，因为这样的列对于本集合内所有的样本都是取同一个值。因为拆分的过程就是对应了一个天然的聚类的过程，activity_category属性对应了聚类的id。因此这里删除该列：

```
types=['type %d'%i for i in range(1,8)]
for _type in types:
    train_datas[_type].drop(['activity_category'],axis=1,inplace=True)
    test_datas[_type].drop(['activity_category'],axis=1,inplace=True)
```

观察丢弃activity_category之后的数据，以train_datas中的type 1 ~3为例，如图 13.7 所示。

In [14]:
```
train_datas['type 1'].head(2)
```
Out[14]:

	activity_id	date_act	activity_category	char_1_act	char_2_act	char_3_act	char_4_act	char_5_act	char_6_act	char_7_act
people_id										
ppl_100025	act1_9923	2022-11-25	type 1	type 3	type 5	type 1	type 1	type 6	type 3	type 3
ppl_100033	act1_198174	2022-07-26	type 1	type 36	type 11	type 5	type 1	type 6	type 1	type 1

2 rows × 53 columns

In [15]:
```
train_datas['type 2'].head(2)
```
Out[15]:

	activity_id	date_act	activity_category	char_10_act	outcome	char_1_people	group_1	char_2_people	date_people	char_
people_id										
ppl_100	act2_2434093	2022-09-27	type 2	type 1	0	type 2	group 17304	type 2	2021-06-29	type 5
ppl_100	act2_3404049	2022-09-27	type 2	type 1	0	type 2	group 17304	type 2	2021-06-29	type 5

2 rows × 45 columns

In [16]:
```
train_datas['type 3'].head(2)
```
Out[16]:

	activity_id	date_act	activity_category	char_10_act	outcome	char_1_people	group_1	char_2_people	date_people	char_
people_id										
ppl_100006	act2_2380649	2022-07-27	type 3	type 114	1	type 2	group 6534	type 3	2022-07-27	type
ppl_100019	act2_1681678	2023-03-26	type 3	type 2	1	type 2	group 45749	type 3	2023-03-26	type

2 rows × 45 columns

图 13.6　train_splits0

In [22]:
```
train_datas['type 1'].head(2)
```
Out[22]:

	activity_id	date_act	char_1_act	char_2_act	char_3_act	char_4_act	char_5_act	char_6_act	char_7_act	char_8_act	...	cha
people_id												
ppl_100025	act1_9923	2022-11-25	type 3	type 5	type 1	type 1	type 6	type 3	type 3	type 6	...	Fal
ppl_100033	act1_198174	2022-07-26	type 36	type 11	type 5	type 1	type 6	type 1	type 1	type 4	...	Fal

2 rows × 52 columns

In [23]:
```
train_datas['type 2'].head(2)
```
Out[23]:

	activity_id	date_act	char_10_act	outcome	char_1_people	group_1	char_2_people	date_people	char_3_people	char_4_
people_id										
ppl_100	act2_2434093	2022-09-27	type 1	0	type 2	group 17304	type 2	2021-06-29	type 5	type 5
ppl_100	act2_3404049	2022-09-27	type 1	0	type 2	group 17304	type 2	2021-06-29	type 5	type 5

2 rows × 44 columns

In [24]:
```
train_datas['type 3'].head(2)
```
Out[24]:

	activity_id	date_act	char_10_act	outcome	char_1_people	group_1	char_2_people	date_people	char_3_people	char_4_
people_id										
ppl_100006	act2_2380649	2022-07-27	type 114	1	type 2	group 6534	type 3	2022-07-27	type 40	type 25
ppl_100019	act2_1681678	2023-03-26	type 2	1	type 2	group 45749	type 3	2023-03-26	type 40	type 25

2 rows × 44 columns

图 13.7　train_splits_drop

13.2.4　去除唯一值

首先将activity_id这一列数据变成列索引。

```
types=['type %d'%i for i in range(1,8)]
for _type in types:
    train_datas[_type].set_index(keys=['activity_id'], drop=True, append=True, inplace=True)
    test_datas[_type].set_index(keys=['activity_id'], drop=True, append=True, inplace=True)
```

这里append=True是保留原来的people_id行索引，从而生成一个多级行索引。inplace=True是原地修改数据。观察数据，以train_datas中的type 1 ~3为例，如图 13.8 所示。

In [27]: `train_datas['type 1'].head(2)`

Out[27]:

		date_act	char_1_act	char_2_act	char_3_act	char_4_act	char_5_act	char_6_act	char_7_act	char_8_act	char_
people_id	activity_id										
ppl_100025	act1_9923	2022-11-25	type 3	type 5	type 1	type 1	type 6	type 3	type 3	type 6	type 8
ppl_100033	act1_198174	2022-07-26	type 36	type 11	type 5	type 1	type 6	type 1	type 1	type 4	type 1

2 rows × 51 columns

In [28]: `train_datas['type 2'].head(2)`

Out[28]:

		date_act	char_10_act	outcome	char_1_people	group_1	char_2_people	date_people	char_3_people	char_4_
people_id	activity_id									
ppl_100	act2_2434093	2022-09-27	type 1	0	type 2	group 17304	type 2	2021-06-29	type 5	type 5
	act2_3404049	2022-09-27	type 1	0	type 2	group 17304	type 2	2021-06-29	type 5	type 5

2 rows × 43 columns

In [29]: `train_datas['type 3'].head(2)`

Out[29]:

		date_act	char_10_act	outcome	char_1_people	group_1	char_2_people	date_people	char_3_people	char_4
people_id	activity_id									
ppl_100006	act2_2380649	2022-07-27	type 114	1	type 2	group 6534	type 3	2022-07-27	type 40	type 25
ppl_100019	act2_1681678	2023-03-26	type 2	1	type 2	group 45749	type 3	2023-03-26	type 40	type 25

2 rows × 43 columns

图 13.8 test_type1_head_resetindex

观察发现，每一行数据对应唯一的一个索引对 (元组(people_id,activit_id))。也就是得到的结果中不包含重复索引。这一点可以通过下面的代码验证：

```
types=['type %d'%i for i in range(1,8)]
for _type in types:
    print(train_datas[_type].index.is_unique,end=',')
    print(test_datas[_type].index.is_unique,end=',' )
```

结果为：

```
True,True,True,True,True,True,True,True,True,True,True,True,True,True,
```

这说明后续训练中不用考虑people_id,activit_id，这是因为它们作为索引，对每个样本是唯一的。它对于判定样本的性质没有任何帮助。这一类属性通常是人工产生的，比如数据库中众多id类的数据，它们通常是作为索引存在的。

13.2.5 数据类型转换

在目前得到的结果中，有不少列的数据都是字符串类型，而且含有文字类型的字符串，如列date_act。因此需要对它进行处理。先查看每个列的数据类型，我们在jupyter notebook中执行：

```
pd.DataFrame({'train_1':train_datas['type 1'].dtypes,
    'train_2':train_datas['type 2'].dtypes,'train_3':train_datas['type 3'].dtypes,
    'train_4':train_datas['type 4'].dtypes, 'train_5':train_datas['type 5'].dtypes,
    'train_6':train_datas['type 6'].dtypes, 'train_7':train_datas['type 7'].dtypes,
    'test_1':test_datas['type 1'].dtypes,'test_2':test_datas['type 2'].dtypes,
    'test_3':test_datas['type 3'].dtypes,'test_4':test_datas['type 4'].dtypes,
    'test_5':test_datas['type 5'].dtypes,'test_6':test_datas['type 6'].dtypes,
    'test_7':test_datas['type 7'].dtypes,})
```

给出部分结果的截图如图 13.9 所示。

	test_1	test_2	test_3	test_4	test_5	test_6	test_7	train_1	train_2
char_10_act	NaN	object	object	object	object	object	object	NaN	object
char_10_people	bool	bool	bool	bool	bool	bool	bool	bool	bool
char_11	bool	bool	bool	bool	bool	bool	bool	bool	bool
char_12	bool	bool	bool	bool	bool	bool	bool	bool	bool
char_13	bool	bool	bool	bool	bool	bool	bool	bool	bool
char_14	bool	bool	bool	bool	bool	bool	bool	bool	bool
char_15	bool	bool	bool	bool	bool	bool	bool	bool	bool
char_16	bool	bool	bool	bool	bool	bool	bool	bool	bool
char_17	bool	bool	bool	bool	bool	bool	bool	bool	bool
char_18	bool	bool	bool	bool	bool	bool	bool	bool	bool
char_19	bool	bool	bool	bool	bool	bool	bool	bool	bool
char_1_act	object	NaN	NaN	NaN	NaN	NaN	NaN	object	NaN

图 13.9 dtypes

其中:type 1数据中没有char_10_act；type 2~7数据中没有char_1_act~char_9_act。train数据中没有outcome。需要将所有的数据转换成np.float64，从而方便后续计算。

观察这些列的规律，发现：

- group_1列为字符串:group xxx，其中x为数字。
- date_act/date_people列为datetime64类型。我们的想法是:将每个日期转换成从1970-01-01日以来的天数，转换为浮点数。
- char_1_act~char_10_act、char_1_people~char_9_people列为字符串：type x：其中x为数字。
- char_10_people、char_11~char_37列为布尔值。将这些列的数据转换成整数 0 和 1。
- outcome、char_38列为整数。其中outcome列为标记信息，其取值范围为 0 ~ 1；char_38列为连续值。

因此执行操作：

```
str_col_list=['group_1']+['char_%d_act'%i for i in range(1,11)]+\
        ['char_%d_people'%i for i in range(1,10)]
```

```
bool_col_list=['char_10_people']+['char_%d'%i for i in range(11,38)]
types=['type %d'%i for i in range(1,8)]
for _type in types:
    for data_set in [train_datas,test_datas]:
        data_set[_type].date_act= (data_set[_type].date_act\
            - np.datetime64('1970-01-01'))/ np.timedelta64(1, 'D')
        data_set[_type].date_people= (data_set[_type].date_people\
            - np.datetime64('1970-01-01'))/ np.timedelta64(1,'D')
        data_set[_type].group_1=data_set[_type].group_1.str.replace("group",'')\
            .str.strip().astype(np.float64)
        for col in bool_col_list:
               if col in data_set[_type]:data_set[_type][col]=\
            data_set[_type][col].astype(np.float64)
        for col in str_col_list[1:]:
               if col in data_set[_type]:data_set[_type][col]=\
            data_set[_type][col].str.replace("type",'').str.strip().astype(np.float64)

        data_set[_type]= data_set[_type].astype(np.float64)
```

这里采用的是Pandas对象的矢量化字符串方法.str.replace()和.str.strp()，它们都返回一个Pandas.Series对象。然后使用Pandas.astype()方法来将数字形式的字符串转换为浮点数。

判断这些数据的类型：

```
types=['type %d'%i for i in range(1,8)]
for _type in types:
    print((train_datas[_type].dtypes==np.float64).all(),end=',')
    print((test_datas[_type].dtypes==np.float64).all(),end=',')
```

结果为：

```
True,True,True,True,True,True,True,True,True,True,True,True,True,True,
```

以train_datas['type 1']为例，我们查看前 5 行数据，如图 13.10 所示。

		date_act	char_1_act	char_2_act	char_3_act	char_4_act	char_5_act	char_6_act	char_7_act	char_8_act	char_9_
people_id	activity_id										
ppl_100025	act1_9923	19321.0	3.0	5.0	1.0	1.0	6.0	3.0	3.0	6.0	8.0
ppl_100033	act1_198174	19199.0	36.0	11.0	5.0	1.0	6.0	1.0	1.0	4.0	1.0
	act1_214090	19523.0	24.0	6.0	6.0	3.0	1.0	3.0	4.0	5.0	1.0
	act1_230588	19416.0	2.0	2.0	3.0	3.0	5.0	2.0	2.0	4.0	2.0
	act1_271874	19199.0	2.0	5.0	3.0	2.0	6.0	1.0	1.0	6.0	8.0

5 rows × 51 columns

图 13.10　train_type1_float

13.2.6　Data_Cleaner 类

根据前面的分析，给出一个 Data_Cleaner类。该类提供了.load_data()方法，返回清洗好的数据。

```
import numpy as np
import pandas as pd
import  pickle
import  time
import os
def current_time():
    '''
    以固定格式打印当前时间

    :return:返回当前时间的字符串
    '''
    return time.strftime('%Y-%m-%d %X', time.localtime())
class Data_Cleaner:
    '''
    数据清洗器

    它的初始化需要提供三个文件的文件名。它提供了唯一的对外接口：load_data()。它返回清洗好的数
        据。
    如果数据已存在，则直接返回。否则将执行一系列清洗操作，并返回清洗好的数据。
    '''
    def __init__(self,people_file_name,act_train_file_name,act_test_file_name):
        '''


        :param people_file_name: people.csv文件的 file_path
        :param act_train_file_name: act_train.csv文件的 file_path
        :param act_test_file_name:act_test.csv文件的 file_path
        :return:
        '''
        self.p_fname=people_file_name
        self.train_fname=act_train_file_name
        self.test_fname=act_test_file_name
        self.types=['type %d'%i for i in range(1,8)]
        self.fname='output/cleaned_data'
    def load_data(self):
        '''
        加载清洗好的数据

        如果数据已经存在，则直接返回。如果不存在，则加载 csv文件，然后合并数据、拆分成
        type1 ~type7，然后执行数据类型转换，最后重新排列每个列的顺序。再然后保存数据并返回数
            据。
```

```
        :return:一个元组：依次为：self.train_datas,self.test_datas
        '''
        if(self._is_ready()):
            print("cleaned data is availiable!\n")
            self._load_data()
        else:
            self._load_csv()
            self._merge_data()
            self._split_data()
            self._typecast_data()
            self._save_data()
        return self.train_datas,self.test_datas

    def _load_csv(self):
        '''
        加载 csv 文件

        :return:
        '''
        print("----- Begin run load_csv at %s -------"%current_time())
        self.people=pd.read_csv(self.p_fname,sep=',',header=0,keep_default_na=True,\
            parse_dates=['date'])
        self.act_train=pd.read_csv(self.train_fname,sep=',',header=0,\
            keep_default_na=True,parse_dates=['date'])
        self.act_test=pd.read_csv(self.test_fname,sep=',',header=0,\
            keep_default_na=True,parse_dates=['date'])

        self.people.set_index(keys=['people_id'],drop=True,append=False,inplace=True)
        self.act_train.set_index(keys=['people_id'],drop=True,append=False,inplace=True)
        self.act_test.set_index(keys=['people_id'],drop=True,append=False,inplace=True)

        print("----- End run load_csv at %s -------"%current_time())
    def _merge_data(self):
        '''
        合并 people 数据和 activity 数据

        :return:
        '''
        print("----- Begin run merge_data at %s -------"%current_time())
        self.train_data=self.act_train.merge(self.people,how='left',left_index=True,\
            right_index=True,suffixes=('_act', '_people'))
        self.test_data=self.act_test.merge(self.people,how='left',left_index=True,\
            right_index=True,suffixes=('_act', '_people'))
        print("----- End run merge_data at %s -------"%current_time())
    def _split_data(self):
```

```
        '''
        拆分数据为 type 1~ 7

        :return:
        '''
        print("----- Begin run split_data at %s -------"%current_time())
        self.train_datas={}
        self.test_datas={}
        for _type in self.types:
            ## 拆分
            self.train_datas[_type]=self.train_data[self.train_data.activity_category==\
            _type].dropna(axis=(0,1), how='all')
            self.test_datas[_type]=self.test_data[self.test_data.activity_category==\
            _type].dropna(axis=(0,1), how='all')
            # 删除列 activity_category
            self.train_datas[_type].drop(['activity_category'],axis=1,inplace=True)
            self.test_datas[_type].drop(['activity_category'],axis=1,inplace=True)
            # 将列 activity_id 作为索引
            self.train_datas[_type].set_index(keys=['activity_id'], drop=True, \
            append=True, inplace=True)
            self.test_datas[_type].set_index(keys=['activity_id'], drop=True,\
            append=True, inplace=True)
        print("----- End run split_data at %s -------"%current_time())

    def _typecast_data(self):
        '''
        执行数据类型转换，将所有数据转换成浮点数

        :return:
        '''
        print("----- Begin run typecast_data at %s -------"%current_time())
        str_col_list=['group_1']+['char_%d_act'%i for i in range(1,11)]+\
            ['char_%d_people'%i for i in range(1,10)]
        bool_col_list=['char_10_people']+['char_%d'%i for i in range(11,38)]

        for _type in self.types:
            for data_set in [self.train_datas,self.test_datas]:
                # 处理日期列
                data_set[_type].date_act= (data_set[_type].date_act-\
            np.datetime64('1970-01-01'))/ np.timedelta64(1, 'D')
                data_set[_type].date_people= (data_set[_type].date_people-\
            np.datetime64('1970-01-01'))/ np.timedelta64(1,'D')
                # 处理 group 列
                data_set[_type].group_1=data_set[_type].group_1.str.replace("group",'').\
            str.strip().astype(np.float64)
```

```
            # 处理布尔值列
            for col in bool_col_list:
                if col in data_set[_type]:data_set[_type][col]=\
        data_set[_type][col].astype(np.float64)
            # 处理其他字符串列
            for col in str_col_list[1:]:
                if col in data_set[_type]:data_set[_type][col]=\
        data_set[_type][col].str.replace("type",'').str.strip().astype(np.float64)

        data_set[_type]= data_set[_type].astype(np.float64)
        print("----- End run typecast_data at %s -------"%current_time())
    def _is_ready(self):
        if(os.path.exists(self.fname)):
            return True
        else :
            return False
    def _save_data(self):
        print("----- Begin run save_data at %s -------"%current_time())
        with open(self.fname,"wb") as file:
            pickle.dump([self.train_datas,self.test_datas],file=file)
        print("----- End run save_data at %s -------"%current_time())
    def _load_data(self):
        print("----- Begin run _load_data at %s -------"%current_time())
        with open(self.fname,"rb") as file:
            self.train_datas,self.test_datas=pickle.load(file)
        print("----- End run _load_data at %s -------"%current_time())
```

13.3 数据预处理

13.3.1 独热码编码

下面考察各列的取值集合。在jupyter notebook中执行下列代码：

```
lambda_len=lambda x:len(x.unique())
lambda_data=lambda x:str(x.unique()) if(len(x.unique())<=3) \
        else str(x.unique()[:3])+'...'
train_results={}
test_results={}
types=['type %d'%i for i in range(1,8)]
for _type in types:
    train_results[_type[-1]]=pd.DataFrame({'len':train_datas[_type].apply(lambda_len),
                    'data':train_datas[_type].apply(lambda_data)},
                    index=train_datas[_type].columns)
    test_results[_type[-1]]=pd.DataFrame({'len':test_datas[_type].apply(lambda_len),
```

```
                    'data':train_datas[_type].apply(lambda_data)},
                    index=test_datas[_type].columns)

train_12=train_results['1'].merge(train_results['2'],how='outer',left_index=True,
        right_index=True,suffixes=('_ta_1', '_ta_2'))
train_34=train_results['3'].merge(train_results['4'],how='outer',left_index=True,
        right_index=True,suffixes=('_ta_3', '_ta_4'))
train_56=train_results['5'].merge(train_results['6'],how='outer',left_index=True,
        right_index=True,suffixes=('_ta_5', '_ta_6'))
train_test_77=train_results['7'].merge(test_results['7'],how='outer',left_index=True,
        right_index=True,suffixes=('_ta_7', '_tt_7'))
test_12=test_results['1'].merge(test_results['2'],how='outer',left_index=True,
        right_index=True,suffixes=('_tt_1', '_tt_2'))
test_34=test_results['3'].merge(test_results['4'],how='outer',left_index=True,
        right_index=True,suffixes=('_tt_3', '_tt_4'))
test_56=test_results['5'].merge(test_results['6'],how='outer',left_index=True
        ,right_index=True,suffixes=('_tt_5', '_tt_6'))

train_12.merge(train_34,how='outer',left_index=True,right_index=True)\
    .merge(train_56,how='outer',left_index=True,right_index=True)  \
    .merge(train_test_77,how='outer',left_index=True,right_index=True)\
    .merge(test_12,how='outer',left_index=True,right_index=True) \
    .merge(test_34,how='outer',left_index=True,right_index=True) \
    .merge(test_56,how='outer',left_index=True,right_index=True)
```

结果截图如图 13.11 所示（其中ta表示训练集train，tt表示测试集test，后缀1表示type 1）。

	data_ta_1	len_ta_1	data_ta_2	len_ta_2	data_ta_3	len_ta_3	data_ta_4	len_ta_4	data_ta_5	len_ta_5	...	data_tt_2	len_tt_
char_10_act	NaN	NaN	[1.]	1.0	[114. 2. 23.]...	450.0	[76. 1727. 894.]...	3315.0	[5493. 489. 584.]...	2747.0	...	[1.]	1.0
char_10_people	[0. 1.]	2.0	[1. 0.]	2.0	[0. 1.]	2.0	[1. 0.]	2.0	[1. 0.]	2.0	...	[1. 0.]	2.0
char_11	[0. 1.]	2.0	[0. 1.]	2.0	[0. 1.]	2.0	[0. 1.]	2.0	[0. 1.]	2.0	...	[0. 1.]	2.0
char_12	[0. 1.]	2.0	[0. 1.]	2.0	[0. 1.]	2.0	[0. 1.]	2.0	[1. 0.]	2.0	...	[0. 1.]	2.0
char_13	[0. 1.]	2.0	[1. 0.]	2.0	[0. 1.]	2.0	[1. 0.]	2.0	[1. 0.]	2.0	...	[1. 0.]	2.0
char_14	[0. 1.]	2.0	[1. 0.]	2.0	[0. 1.]	2.0	[1. 0.]	2.0	[1. 0.]	2.0	...	[1. 0.]	2.0
char_15	[0. 1.]	2.0	[0. 1.]	2.0	[0. 1.]	2.0	[0. 1.]	2.0	[1. 0.]	2.0	...	[0. 1.]	2.0
char_16	[0. 1.]	2.0	[1. 0.]	2.0	[0. 1.]	2.0	[1. 0.]	2.0	[1. 0.]	2.0	...	[1. 0.]	2.0
char_17	[0. 1.]	2.0	[0. 1.]	2.0	[0. 1.]	2.0	[0. 1.]	2.0	[1. 0.]	2.0	...	[0. 1.]	2.0
char_18	[0. 1.]	2.0	[0. 1.]	2.0	[0. 1.]	2.0	[0. 1.]	2.0	[1. 0.]	2.0	...	[0. 1.]	2.0
char_19	[0. 1.]	2.0	[0. 1.]	2.0	[0. 1.]	2.0	[0. 1.]	2.0	[1. 0.]	2.0	...	[0. 1.]	2.0
char_1_act	[3. 36. 24.]...	51.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN
char_1_people	[2. 1.]	2.0	[2. 1.]	2.0	[2. 1.]	2.0	[2. 1.]	2.0	[2. 1.]	2.0		[2. 1.]	2.0

图 13.11 data_content

我们发现：

- ❑ 对于char_1_act~char_9_act列，这些列将使用独热码编码。

- ❑ 对于char_1_people~char_9_people列，这些列将使用独热编码。
- ❑ 对于char_10_people和char_11~char_37列，这些列取值集合为{0,1}，可以不用编码。
- ❑ date_act和date_people是时间量，它们已经转换成了距离1970-01-01日以来的天数，不需要编码。
- ❑ char_38是连续值，不需要编码。
- ❑ outcome为监督信息，该列也不需要编码。
- ❑ group_1和char_10_act列理论上需要进行独热码编码。但是考虑到这些列的取值集合非常大，一旦进行独热码编码，则样本的特征数量将呈爆炸式增长。因此为了计算方便，并不对该列进行独热码编码。

> 如果对group_1和char_10_act列进行独热码编码，则各个数据集的样本的特征数量都在 10000 的量级。

为了计算方便，将被编码的列char_1_act~char_9_act,char_1_people~char_9_people放在最前面，将group_1,char_10_act,date_act,date_people,char_38,outcome列放到最末尾。给出编码的代码如下：

```
from scipy.sparse import hstack,csr_matrix
from sklearn.preprocessing  import OneHotEncoder
def onehot_encode(train_datas,test_datas):

    train_results={}
    test_results={}
    types=['type %d'%i for i in range(1,8)]
    for _type in types:
        if _type=='type 1':
            one_hot_cols=['char_%d_act'%i for i in range(1,10)]+\
            ['char_%d_people'%i for i in range(1,10)]
            train_end_cols=['group_1','date_act','date_people','char_38','outcome']
            test_end_cols=['group_1','date_act','date_people','char_38']
        else:
            one_hot_cols=['char_%d_people'%i for i in range(1,10)]
            train_end_cols=['group_1','char_10_act','date_act','date_people',\
            'char_38','outcome']
            test_end_cols=['group_1','char_10_act','date_act',\
            'date_people','char_38']

        train_front_array=train_datas[_type][one_hot_cols].values #头部数组
        train_end_array=train_datas[_type][train_end_cols].values#末尾数组
        train_middle_array=train_datas[_type].drop(train_end_cols\
            +one_hot_cols,axis=1,inplace=False).values#中间数组

        test_front_array=test_datas[_type][one_hot_cols].values #头部数组
```

```
        test_end_array=test_datas[_type][test_end_cols].values#末尾数组
        test_middle_array=test_datas[_type].drop(test_end_cols\
            +one_hot_cols,axis=1,inplace=False).values#中间数组

        encoder=OneHotEncoder(categorical_features='all',sparse=True)
        # 一个稀疏矩阵，类型为 csr_matrix
        train_result=hstack([encoder.fit_transform(train_front_array),\
            csr_matrix(train_middle_array),csr_matrix(train_end_array)])
        test_result=hstack([encoder.transform(test_front_array),\
            csr_matrix(test_middle_array),csr_matrix(test_end_array)])
        train_results[_type]=train_result
        test_results[_type]=test_result
    return train_results,test_results
```

在jupyter notebook中调用onehot_encode：

```
types=['type %d'%i for i in range(1,8)]

print('before encode:\n')
for _type in types:
    print('train(type=%s):shape='%_type,train_datas[_type].shape)
    print('test(type=%s):shape='%_type,test_datas[_type].shape)
print('==============\n\n')
train_results,test_results=onehot_encode(train_datas,test_datas)
print('after encode:\n')
for _type in types:
    print('train(type=%s):shape='%_type,train_results[_type].shape)
    print('test(type=%s):shape='%_type,test_results[_type].shape)
print('==============\n\n')
```

结果如下：

```
before encode:

train(type=type 1):shape= (157615, 51)
test(type=type 1):shape= (40092, 50)
train(type=type 2):shape= (904683, 43)
test(type=type 2):shape= (223164, 42)
train(type=type 3):shape= (429408, 43)
test(type=type 3):shape= (59931, 42)
train(type=type 4):shape= (207465, 43)
test(type=type 4):shape= (50215, 42)
train(type=type 5):shape= (490710, 43)
test(type=type 5):shape= (123463, 42)
train(type=type 6):shape= (4253, 43)
test(type=type 6):shape= (1051, 42)
train(type=type 7):shape= (3157, 43)
```

```
test(type=type 7):shape= (771, 42)
==============


after encode:

train(type=type 1):shape= (157615, 321)
test(type=type 1):shape= (40092, 320)
train(type=type 2):shape= (904683, 165)
test(type=type 2):shape= (223164, 164)
train(type=type 3):shape= (429408, 164)
test(type=type 3):shape= (59931, 163)
train(type=type 4):shape= (207465, 164)
test(type=type 4):shape= (50215, 163)
train(type=type 5):shape= (490710, 163)
test(type=type 5):shape= (123463, 162)
train(type=type 6):shape= (4253, 155)
test(type=type 6):shape= (1051, 154)
train(type=type 7):shape= (3157, 161)
test(type=type 7):shape= (771, 160)
==============
```

可以看到，经过独热码编码之后，样本的特征数量大量增长。

13.3.2 归一化处理

我们需要对group_1、char_10_act、date_act、date_people、char_38数据进行归一化处理。接着给出归一化处理函数：

```
from sklearn.preprocessing  import MaxAbsScaler
def scale(train_datas,test_datas):
    train_results={}
    test_results={}
    types=['type %d'%i for i in range(1,8)]

    for _type in types:
        if _type=='type 1':
            train_last_index=5#最后5列为 group_1/date_act/date_people/char_38/outcome
            test_last_index=4#最后4列为 group_1/date_act/date_people/char_38
        else:
            train_last_index=6#最后6列为 group_1/char_10_act/date_act/date_people/\
    char_38/outcome
            test_last_index=5#最后5列为 group_1/char_10_act/date_act/date_people/char_38

        scaler=MaxAbsScaler()
```

```
        train_array=train_datas[_type].toarray()
        train_front=train_array[:,:-train_last_index]
        train_mid=scaler.fit_transform(train_array[:,-train_last_index:-1])\
            #outcome 不需要归一化
        train_end=train_array[:,-1].reshape((-1,1)) #outcome
        train_results[_type]=np.hstack((train_front,train_mid,train_end))

        test_array=test_datas[_type].toarray()
        test_front=test_array[:,:-test_last_index]
        test_end=scaler.transform(test_array[:,-test_last_index:])
        test_results[_type]=np.hstack((test_front,test_end))

    return train_results,test_results
```

由于之前已经将这些列交换到了所有列的末尾，因此这里直接分片选取即可。检验归一化之后的结果，检验代码如下：

```
ta_results,tt_results=scale(train_results,test_results)
types=['type %d'%i for i in range(1,8)]
for _type in types:
    print("Train(type=%s):"%_type,np.unique(ta_results[_type].max(axis=1)),\
            np.unique(ta_results[_type].min(axis=1)))
    print("Test(type=%s):"%_type,np.unique(tt_results[_type].max(axis=1)),\
            np.unique(tt_results[_type].min(axis=1)))
```

结果如下：

```
Train(type=type 1): [ 1.] [ 0.]
Test(type=type 1): [ 1.] [ 0.]
Train(type=type 2): [ 1.] [ 0.]
Test(type=type 2): [ 1.] [ 0.]
Train(type=type 3): [ 1.] [ 0.]
Test(type=type 3): [ 1.] [ 0.]
Train(type=type 4): [ 1.] [ 0.]
Test(type=type 4): [ 1.] [ 0.]
Train(type=type 5): [ 1.] [ 0.]
Test(type=type 5): [ 1.] [ 0.]
Train(type=type 6): [ 1.] [ 0.]
Test(type=type 6): [ 1.] [ 0.]
Train(type=type 7): [ 1.] [ 0.]
Test(type=type 7): [ 1.] [ 0.]
```

可以看到，归一化之后，所有数据集的所有列都归一化到[0,1]。

13.3.3 Data_Preprocesser 类

这里并没有给出特征提取的过程，因为后面将使用GBDT模型来学习。在GBDT模型中，有一个参数，该参数可以选取部分特征来学习。这个过程同时对应了特征提取的过程。

综合前面的分析，给出Data_Preprocesser类。该类提供了.load_data()方法，返回预处理好的数据。

```
import  os
import  pickle
import numpy as np
from scipy.sparse import hstack,csr_matrix
from sklearn.preprocessing  import OneHotEncoder,MaxAbsScaler
from data_clean import current_time
from data_clean import Data_Cleaner


class Data_Preprocesser:
    '''
    数据预处理器

    它的初始化需要提供清洗好的数据。它提供了唯一的对外接口：load_data()。它返回预处理好的数
        据。
    如果数据已存在，则直接返回。否则将执行一系列预处理操作并返回预处理好的数据。
    '''
    def __init__(self,train_datas,test_datas):
        '''
        :param train_datas: 清洗好的训练集
        :param test_datas: 清洗好的测试集
        :return:
        '''
        self.types=train_datas.keys()
        self.train_datas=train_datas
        self.test_datas=test_datas

        self.fname='output/processed_data'
    def load_data(self):
        '''
        加载预处理好的数据

        如果数据已经存在，则直接返回。如果不存在，预处理数据，并且存储之后返回。

        :return:一个元组：依次为：train_datas,test_datas
        '''
        if(self._is_ready()):
            print("preprocessed data is availiable!\n")
            self._load_data()
```

```
        else:
            self._onehot_encode()
            self._scaled()
            self._save_data()
        return self.train_datas,self.test_datas
    def _onehot_encode(self):
        '''
        独热码编码

        :return:
        '''
        print("----- Begin run onehot_encode at %s -------"%current_time())
        train_results={}
        test_results={}
        self.encoders={}

        for _type in self.types:
            if _type=='type 1':
                one_hot_cols=['char_%d_act'%i for i in range(1,10)]+\
            ['char_%d_people'%i for i in range(1,10)]
                train_end_cols=['group_1','date_act','date_people','char_38','outcome']
                test_end_cols=['group_1','date_act','date_people','char_38']
            else:
                one_hot_cols=['char_%d_people'%i for i in range(1,10)]
                train_end_cols=['group_1','char_10_act','date_act',\
                'date_people','char_38','outcome']
                test_end_cols=['group_1','char_10_act','date_act',\
            'date_people','char_38']

            train_front_array=self.train_datas[_type][one_hot_cols].values #头部数组
            train_end_array=self.train_datas[_type][train_end_cols].values#末尾数组
            train_middle_array=self.train_datas[_type].drop(train_end_cols\
                +one_hot_cols,axis=1,inplace=False).values#中间数组

            test_front_array=self.test_datas[_type][one_hot_cols].values #头部数组
            test_end_array=self.test_datas[_type][test_end_cols].values#末尾数组
            test_middle_array=self.test_datas[_type].drop(test_end_cols\
                +one_hot_cols,axis=1,inplace=False).values#中间数组

            encoder=OneHotEncoder(categorical_features='all',sparse=True)\
            # 一个稀疏矩阵，类型为 csr_matrix
            train_result=hstack([encoder.fit_transform(train_front_array),\
                csr_matrix(train_middle_array),csr_matrix(train_end_array)])
            test_result=hstack([encoder.transform(test_front_array),\
                csr_matrix(test_middle_array),csr_matrix(test_end_array)])
```

```
            train_results[_type]=train_result
            test_results[_type]=test_result
            self.encoders[_type]=encoder

        self.train_datas=train_results
        self.test_datas=test_results
        print("----- End run onehot_encode at %s -------"%current_time())
    def _scaled(self):
        '''
        特征归一化，采用 MaxAbsScaler 来进行归一化
        :return:
        '''
        print("----- Begin run scaled at %s -------"%current_time())
        train_scales={}
        test_scales={}
        self.scalers={}
        for _type in self.types:
            if _type=='type 1':
                    train_last_index=5\
            #最后5列为 group_1/date_act/date_people/char_38/outcome
                    test_last_index=4#最后4列为 group_1/date_act/date_people/char_38
            else:
                    train_last_index=6\
            #最后6列为 group_1/char_10_act/date_act/date_people/char_38/outcome
                    test_last_index=5\
            #最后5列为 group_1/char_10_act/date_act/date_people/char_38

            scaler=MaxAbsScaler()
            train_array=self.train_datas[_type].toarray()
            train_front=train_array[:,:-train_last_index]
            train_mid=scaler.fit_transform(train_array[:,-train_last_index:-1])\
        #outcome 不需要归一化
            train_end=train_array[:,-1].reshape((-1,1)) #outcome
            train_scales[_type]=np.hstack((train_front,train_mid,train_end))

            test_array=self.test_datas[_type].toarray()
            test_front=test_array[:,:-test_last_index]
            test_end=scaler.transform(test_array[:,-test_last_index:])
            test_scales[_type]=np.hstack((test_front,test_end))
            self.scalers[_type]=scaler
        self.train_datas=train_scales
        self.test_datas=test_scales
        print("----- End run scaled at %s -------"%current_time())
    def _is_ready(self):
        if(os.path.exists(self.fname)):
```

```
            return True
        else :
            return False
    def _save_data(self):
        print("----- Begin run save_data at %s -------"%current_time())
        with open(self.fname,'wb') as file:#保存训练集、测试集、编码器、归一化器
            pickle.dump([self.train_datas,self.test_datas,\
            self.encoders,self.scalers],file)
        print("----- End run save_data at %s -------"%current_time())
    def _load_data(self):
        print("----- Begin run _load_data at %s -------"%current_time())
        with open(self.fname,'rb') as file:#加载训练集、测试集、编码器、归一化器
            self.train_datas,self.test_datas,self.encoders,\
            self.scalers=pickle.load(file)
        print("----- End run _load_data at %s -------"%current_time())
```

13.4 学习曲线和验证曲线

13.4.1 程序说明

导入包：

```
import  numpy as np
import  matplotlib.pyplot as plt
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import learning_curve,validation_curve

from sklearn.model_selection import train_test_split
from data_clean import current_time
from data_preprocess import Data_Preprocesser,Data_Cleaner
```

首先给出一个辅助类Curver_Helper。该类的作用是：

- 通过save_curve方法来保存训练结果曲线的坐标点。因为我们通常是在服务器上运行算法，很可能服务器没有桌面环境，因此无法实时绘制曲线。因此我们的想法是将训练结果曲线的数据点保存下来，然后在本地机器上绘制曲线。
- 通过plot_curve方法来实时绘制结果曲线。这一功能通常是在有桌面环境的机器上使用。
- 通过plot_from_saved_data类方法从本地机器上加载结果曲线的数据点，然后绘制曲线。

```
class Curver_Helper:
    '''
    学习曲线和验证曲线的辅助类，用于保存曲线和绘制曲线

    '''
    def __init__(self,curve_name,xlabel,x_islog):
        '''
        初始化函数

        :param curve_name:曲线名称
        :param xlabel:曲线 X轴的名称
        :param x_islog:曲线 X轴是否为对数
        :return:
        '''
        self.curve_name=curve_name
        self.xlabel=xlabel
        self.x_islog=x_islog
    def save_curve(self,x_data,train_scores_mean,train_scores_std,\
            test_scores_mean,test_scores_std):
        '''
        保存曲线的数据

        :param x_data: 曲线的 x 轴数据，也就是被考察的指标的序列
        :param train_scores_mean: 训练集预测的平均得分
        :param train_scores_std:训练集预测得分的标准差
        :param test_scores_mean:测试集预测的平均得分
        :param test_scores_std:测试集预测得分的标准差
        :return:
        '''
        with open("output/%s"%self.curve_name,"wb") as output:
            result_array=np.array([x_data,train_scores_mean,\
            train_scores_std,test_scores_mean,test_scores_std])
            np.save(output,result_array)
    def plot_curve(self,x_data,train_scores_mean,train_scores_std
                   ,test_scores_mean,test_scores_std):
            '''
            绘图并保存图片

            :param x_data:曲线的 x 轴数据，也就是被考察的指标的序列
            :param train_scores_mean:训练集预测的平均得分
            :param train_scores_std:训练集预测得分的标准差
            :param test_scores_mean:测试集预测的平均得分
            :param test_scores_std:测试集预测得分的标准差
            :return:
            '''
```

```
        min_y1=np.min(train_scores_mean)
        min_y2=np.min(test_scores_mean)
        fig=plt.figure(figsize=(20,15))
        ax=fig.add_subplot(1,1,1)
        ax.plot(x_data, train_scores_mean, label="Training roc_auc",\
        color="r",marker='o')
        ax.fill_between(x_data, train_scores_mean - train_scores_std,
        train_scores_mean + train_scores_std, alpha=0.2, color="r")
        ax.plot(x_data, test_scores_mean, label="Testing roc_auc", \
        color="g",marker='+')
        ax.fill_between(x_data, test_scores_mean - test_scores_std,
        test_scores_mean + test_scores_std, alpha=0.2, color="g")
        ax.set_title("%s"%self.curve_name)
        ax.set_xlabel("%s"%self.xlabel)
        ax.locator_params(axis='x', tight=True, nbins=10)
        ax.grid(which="both")
        if self.x_islog:ax.set_xscale('log')
        ax.set_ylabel("Score")
        ax.set_ylim(min(min_y1,min_y2)-0.1,1.1)
        ax.set_xlim(0,max(x_data))
        ax.legend(loc='best')
        ax.grid(True,which='both',axis='both')
        fig.savefig("output/%s.png"%self.curve_name,dpi=100)
@classmethod
def plot_from_saved_data(self,file_name,curve_name,xlabel,x_islog):
    '''
    通过保存的数据点来绘制并保存图形

    :param file_name: 保存数据点的文件名
    :param curve_name:曲线名称
    :param xlabel:曲线 X轴的名称
    :param x_islog:曲线 X轴是否为对数
    :return:
    '''
    x_data,train_scores_mean,train_scores_std,test_scores_mean,\
        test_scores_std=np.load(file_name)
    helper=Curver_Helper(curve_name,xlabel,x_islog)
    helper.plot_curve(x_data,train_scores_mean,train_scores_std,\
        test_scores_mean,test_scores_std)
```

我们提供一个辅助函数cut_data。该函数的作用是：切分训练集，使用一部分训练集来训练。通常训练集样本点非常庞大，我们期望通过观察学习曲线来获取一个合适的比例（比如说 30%），从而使用这个比例的训练集来训练数据。这个比例点的选取，通常是在学习曲线上一个平缓的区域取得的。

```
def cut_data(data,scale_factor,stratify=True,seed=0):
    '''
    切分数据集，使用其中的一部分来学习

    :param data:原始数据集
    :param scale_factor:传递给 train_test_split 的 train_size 参数，\
        可以为浮点数([0,1.0])，可以为整数
    :param stratify:传递给 train_test_split 的 stratify 参数
    :param seed: 传递给 train_test_split 的 seed 参数
    :return: 返回一部分数据集
    '''
    if stratify:
            return train_test_split(data,train_size=scale_factor,\
            stratify=data[:,-1],random_state=seed)[0]
    else:
            return train_test_split(data,train_size=scale_factor,\
            random_state=seed)[0]
```

提供学习曲线和验证曲线的类为LearningCurver类和ValidationCurver类。这两个类提供的唯一接口为create_curve方法。

```
class Curver:
    '''
    用于生成学习曲线验证曲线的父类
    '''
    def create_curve(self,train_data,curve_name,xlabel,x_islog,scale=0.1,is_gui=False):
        '''
        生成曲线

        :param train_data:训练数据集
        :param curve_name : 曲线名字，用于绘图和保存文件
        :param xlabel: 曲线 X轴名字
        :param x_islog: X轴是否为对数坐标
        :param scale:切分比例，默认使用 10%的训练集
        :param is_gui:是否在 GUI环境下。如果在 GUI环境下，则绘制图片并保存
        :return:
        '''
        class_name=self.__class__.__name__
        self.curve_name=curve_name
        ###  加载数据
        data=cut_data(train_data,scale_factor=scale,stratify=True,seed=0)
        self.X=data[:,:-1]
        self.y=data[:,-1]
        ##### 生成曲线参数 ###
        result=self._curve()
        ####### 保存和绘制曲线 #######
```

```
        self.helper=Curver_Helper(self.curve_name,xlabel,x_islog)
        if(is_gui):self.helper.plot_curve(*result)
        self.helper.save_curve(*result)
class LearningCurver(Curver):
    def __init__(self,train_sizes):
        self.train_sizes=train_sizes
        self.estimator=GradientBoostingClassifier(max_depth=10)

    def _curve(self):
        print("----- Begin run learning_curve(%s) at %s -------"%\
            (self.curve_name,current_time()))
        #### 获取学习曲线 ######
        abs_trains_sizes,train_scores, test_scores = learning_curve(self.estimator,\
                self.X, self.y,cv=3,scoring="roc_auc",train_sizes=self.train_sizes,\
            n_jobs=-1,verbose=1)
        print("----- End run learning_curve(%s) at %s -------"%\
            (self.curve_name,current_time()))
        ###### 对每个 test_size ，获取 3 折交叉上预测得分的均值和方差 #####
        train_scores_mean = np.mean(train_scores, axis=1)
        train_scores_std = np.std(train_scores, axis=1)
        test_scores_mean = np.mean(test_scores, axis=1)
        test_scores_std = np.std(test_scores, axis=1)
        return abs_trains_sizes,train_scores_mean,train_scores_std,\
            test_scores_mean,test_scores_std
class ValidationCurver(Curver):
    def __init__(self,param_name,param_range):
        self.p_name=param_name
        self.p_range=param_range
        self.estimator=GradientBoostingClassifier()
    def _curve(self):
        print("----- Begin run validation_curve(%s) at %s -------"%\
            (self.curve_name,current_time()))
        train_scores, test_scores = validation_curve(self.estimator, self.X, self.y,\
             param_name=self.p_name,param_range=self.p_range,\
            cv=3, scoring="roc_auc",n_jobs=-1,verbose=1)
        print("----- End run validation_curve(%s) at %s -------"%\
            (self.curve_name,current_time()))
        train_scores_mean = np.mean(train_scores, axis=1)
        train_scores_std = np.std(train_scores, axis=1)
        test_scores_mean = np.mean(test_scores, axis=1)
        test_scores_std = np.std(test_scores, axis=1)
        return [item for item in self.p_range],train_scores_mean,train_scores_std,\
            test_scores_mean,test_scores_std
```

最后提供以下几个辅助方法：

- run_learning_curve用于生成学习曲线。
- run_test_subsample用于产生subsample参数的验证曲线。其中scale参数从学习曲线中获取。
- run_test_n_estimators用于产生n_estimators参数的验证曲线。其中scale参数从学习曲线中获取，subsample参数从subsample参数的验证曲线中获取。
- run_test_maxdepth用于产生maxdepth参数的验证曲线。其中scale参数从学习曲线中获取，subsample参数从subsample参数的验证曲线中获取，n_estimators参数从n_estimators参数的验证曲线中获取。

```
def run_learning_curve(data,type_name):
    '''
    生成学习曲线

    :param data: 训练集
    :param type_name: 数据种类名
    :return:
    '''
    learning_curver=LearningCurver(train_sizes=\
        np.logspace(-1,0,num=10,endpoint=True,dtype='float'))
    learning_curver.create_curve(data,"learning_curve_%s"%type_name,\
    xlabel="Nums",x_islog=True,scale=0.99,is_gui=True)
def run_test_subsample(data,type_name,scale,param_range):
    '''
    生成验证曲线，验证 subsample 参数

    :param data: 训练集
    :param type_name: 数据种类名
    :param scale: 样本比例，一个小于1.0的浮点数
    :param param_range: subsample 参数的范围
    :return:
    '''
    validation_curver=ValidationCurver("subsample",param_range=param_range)
    validation_curver.create_curve(data,"validation_curve_subsample_%s"%type_name,\
                xlabel='subsample',x_islog=False,scale=scale,is_gui=True)
def run_test_n_estimators(data,type_name,scale,subsample,param_range):
    '''
    生成验证曲线，验证 n_estimators 参数

    :param data: 训练集
    :param type_name: 数据种类名
    :param scale: 样本比例，一个小于1.0的浮点数
    :param subsample: subsample参数
    :param param_range: n_estimators 参数的范围
```

```
    :return:
    '''

    validation_curver=ValidationCurver("n_estimators",param_range=param_range)
    validation_curver.estimator.set_params(subsample=subsample)#调整 subsample
    validation_curver.create_curve(data,"validation_curve_n_estimators_%s"%type_name\
                 ,xlabel='n_estimators',x_islog=True,scale=scale,is_gui=True)
def run_test_maxdepth(data,type_name,scale,subsample,n_estimators,param_range):
    '''
    生成验证曲线，验证 maxdepth 参数

    :param data: 训练集
    :param type_name：数据种类名
    :param scale: 样本比例，一个小于1.0的浮点数
    :param subsample: subsample参数
    :param n_estimators: n_estimators 参数
    :param param_range: maxdepth 参数的范围
    :return:
    '''
    validation_curver=ValidationCurver("max_depth",param_range=param_range)
    validation_curver.estimator.set_params(subsample=subsample) # 调整 subsample
    validation_curver.estimator.set_params(n_estimators=n_estimators) # 调整 n_estimators
    validation_curver.create_curve(data,"validation_curve_maxdepth_%s"%type_name\
             ,xlabel='maxdepth',x_islog=True,scale=scale,is_gui=True)
```

13.4.2 运行结果

我们以type 7为例，因为这种类型的数据比较少，适合于在单机上运行（其他类型的数据可能需要在大型服务器上运行数小时甚至数天）。首先加载数据：

```
clearner=Data_Cleaner("./data/people.csv",'./data/act_train.csv','./data/act_test.csv')
result=clearner.load_data()
preprocessor=Data_Preprocesser(*result)
train_datas,test_datas=preprocessor.load_data()
```

第一步：生成学习曲线：

```
run_learning_curve(train_datas['type 7'],'type7')
```

结果如图 13.12 所示。每个点对应的 x 坐标依次为[208 268 347 448 579 748 966 1248 1612 2082]。可以看到在样本点为966以后，测试集的预测性能（由AUC得分指示）变化缓慢。因此后续将采用966/2082=46.4%比例的训练集来训练，即仅用约 50% 的训练集来训练数据。这里使用AUC而不采用预测准确率 (accurancy)，是考虑到二类分类问题的样本不平衡的缘故。

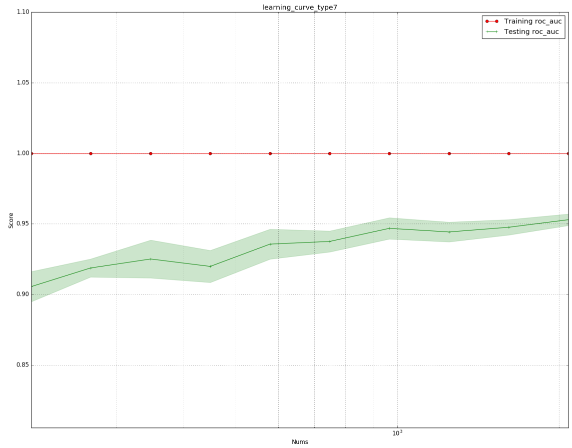


图 13.12 learning_curve_type7

> 通常这是一种预测性能与训练代价的折中。训练集的规模越大，则预测性能越好，但是训练代价越高。训练代价体现在学习模型的时间和硬件资源上。

第二步：生成subsample参数的验证曲线，这里scale=0.5（并没有严格地等于46.4%，而是取一个比它稍大的一个数值）：

```
run_test_subsample(train_datas['type 7'],'type7',0.5,\
        param_range=np.linspace(0.01,1,num=10,dtype=float))
```

结果如图 13.13 所示。每个点对应的 x 坐标依次为[0.01, 0.12, 0.23, 0.34, 0.45, 0.56, 0.67, 0.78, 0.89, 1.]。可以看到在subsample=0.34之后，测试集的预测性能变化缓慢。因此后续将随机采用 34% 的特征来训练数据，即只有大约 40% 的特征。这就是一种数据降维方法。

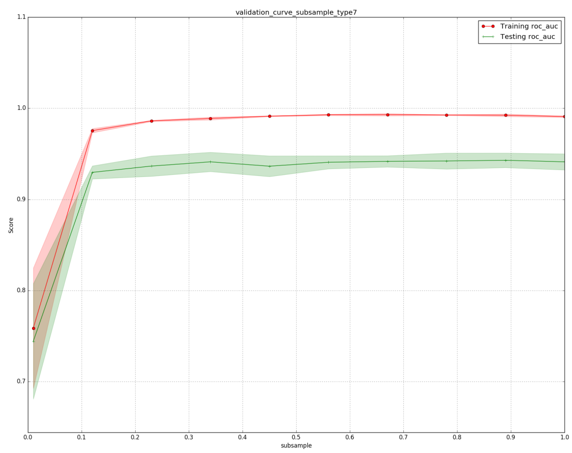


图 13.13 validation_curve_subsample_type7

第三步：生成n_estimators参数的验证曲线。其中scale=0.5，subsample=0.4

```
run_test_n_estimators(train_datas['type 7'],'type7',\
    0.5,0.4,param_range=np.logspace(0,3.5,num=10,dtype=int))
```

结果如图 13.14 所示。每个点对应的x坐标依次为[1,   2,   5,   14,   35,   87, 215, 527, 1291, 3162]。可以看到在n_estimators=35之后，测试集的预测性能变化缓慢。因此后续将使用n_estimators=35来训练数据。

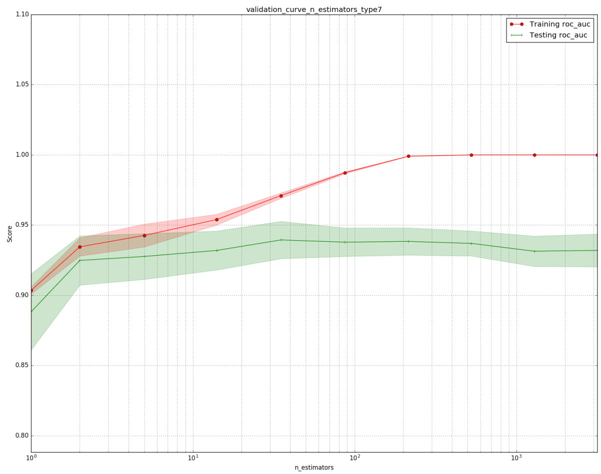


图 13.14 validation_curve_n_estimators_type7

第四步：生成maxdepth参数的验证曲线。其中scale=0.5，subsample=0.4，n_estimators=35：

```
run_test_maxdepth(train_datas['type 7'],'type7',0.5,0.4,35,\
        param_range=np.logspace(0,3,num=10,dtype=int))
```

结果如图 13.15 所示。每个点对应的 x 坐标依次为[1,   2,   4,   10,   21,   46, 100, 215, 464, 1000]。可以看到这里并没有明显的趋势，而是呈现一种波动。

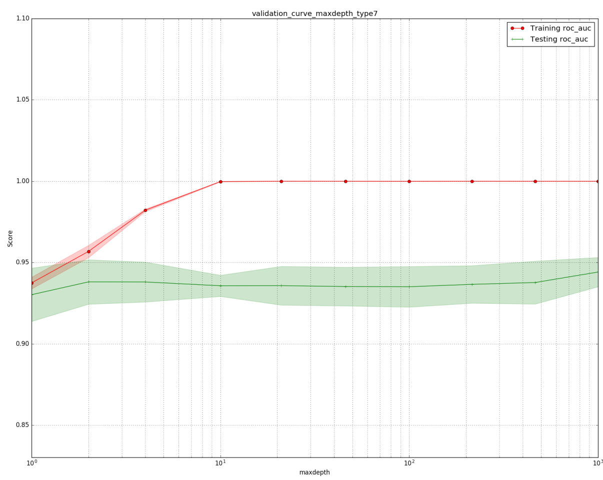


图 13.15 validation_curve_maxdepth_type7

13.5 参数优化

经过了前面的全局性的参数分析，下面给出参数优化的函数：

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report
from data_clean import current_time
from sklearn.model_selection import train_test_split
from data_preprocess import Data_Preprocesser,Data_Cleaner

def grid_search(tuned_parameters,data,train_size,seed):
    '''
    参数优化

    :param tuned_parameters: 待优化的参数字典
    :param data: 数据集
    :param train_size:训练集大小
    :param seed:用于生成随机数种子
    :return:
    '''
    print("----- Begin run grid_search at %s -------"%current_time())
    X=data[:,:-1]
    y=data[:,-1]
    X_train,X_test,y_train,y_test=train_test_split(X,y,train_size=train_size,\
            stratify=data[:,-1],random_state=seed)
    clf=GridSearchCV(GradientBoostingClassifier(),tuned_parameters,cv=10,\
            scoring="roc_auc")
    clf.fit(X_train,y_train)
    print("Best parameters set found:",clf.best_params_)
    print("Randomized Grid scores:")
    for params, mean_score, scores in clf.grid_scores_:
        print("\t%0.3f (+/-%0.03f) for %s" % (mean_score, scores.std() * 2, params))
        print("Optimized Score:",clf.score(X_test,y_test))
        print("Detailed classification report:")
        y_true, y_pred = y_test, clf.predict(X_test)
        print(classification_report(y_true, y_pred))
    print("----- End run grid_search at %s -------"%current_time())
```

给出调用过程：

```
clearner=Data_Cleaner("./data/people.csv",'./data/act_train.csv','./data/act_test.csv')
result=clearner.load_data()
preprocessor=Data_Preprocesser(*result)
train_datas,test_datas=preprocessor.load_data()
tuned_parameters={'subsample':[0.3,0.35,0.4,0.45,0.5,0.55,0.6],
```

```
                    'n_estimators':[5,10,20,40,100],
                    'max_depth':[2,4,8,16,32]}
grid_search(tuned_parameters,train_datas['type 7'],train_size=0.75,seed=0)
```

给出被优化的参数为：

- ❑ subsample：取值集合为[0.3,0.35,0.4,0.45,0.5,0.55,0.6]。
- ❑ n_estimators：取值集合为[30,35,50,100,150,200]。
- ❑ max_depth：取值集合为[2,4,8,16,32]。

这些取值集合的确定的依据，就是我们前面学习曲线和验证曲线的结果。运行的结果如下（因为篇幅的限制，这里我们只是给出了部分结果）：

```
Best parameters set found: {'max_depth': 32, 'subsample': 0.6, 'n_estimators': 50}
Randomized Grid scores:
    0.943 (+/-0.021) for {'max_depth': 2, 'subsample': 0.3, 'n_estimators': 30}
Optimized Score: 0.956403888266
Detailed classification report:
             precision    recall  f1-score   support


        0.0       0.95      0.86      0.90       474
        1.0       0.81      0.93      0.87       316

avg / total       0.89      0.88      0.89       790
...
    0.954 (+/-0.024) for {'max_depth': 32, 'subsample': 0.6, 'n_estimators': 50}
Optimized Score: 0.956403888266
Detailed classification report:
             precision    recall  f1-score   support


        0.0       0.95      0.86      0.90       474
        1.0       0.81      0.93      0.87       316

avg / total       0.89      0.88      0.89       790

    0.948 (+/-0.026) for {'max_depth': 32, 'subsample': 0.3, 'n_estimators': 100}
Optimized Score: 0.956403888266
Detailed classification report:
             precision    recall  f1-score   support


        0.0       0.95      0.86      0.90       474
        1.0       0.81      0.93      0.87       316

avg / total       0.89      0.88      0.89       790
...
```

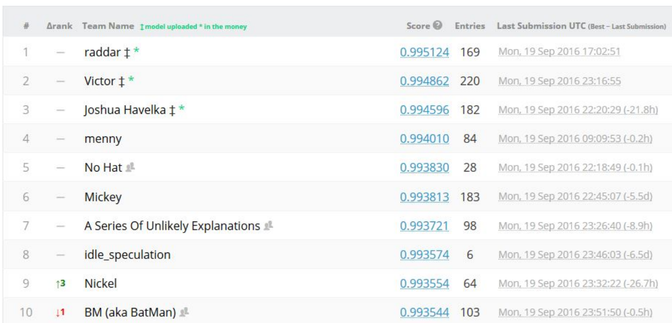
可以看到最优的参数为：subsample=0.6,n_estimators=50,max_depth=32，测试集的平均AUC为0.9564。

13.6 小结

在拆分步骤中，得到了各类型数据的数量如下：

```
type 2    904683
type 5    490710
type 3    429408
type 4    207465
type 1    157615
type 6      4253
type 7      3157
```

以type 7为例，在PC上给出了一个完整的流程示范，有条件的同学可以在高性能服务器上运行所有类型的数据集。大家可以将自己的结果与官方给出的排名进行比较，官方排名结果的网站为：https://www.Kaggle.com/c/predicting-red-hat-business-value/leaderboard。前 10 名的排名截图如图 13.16 所示。



#	Δrank	Team Name ‡model uploaded * in the money	Score	Entries	Last Submission UTC (Best – Last Submission)
1	—	raddar ‡ *	0.995124	169	Mon, 19 Sep 2016 17:02:51
2	—	Victor ‡ *	0.994862	220	Mon, 19 Sep 2016 23:16:55
3	—	Joshua Havelka ‡ *	0.994596	182	Mon, 19 Sep 2016 22:20:29 (-21.8h)
4	—	menny	0.994010	84	Mon, 19 Sep 2016 09:09:53 (-0.2h)
5	—	No Hat	0.993830	28	Mon, 19 Sep 2016 22:18:49 (-0.1h)
6	—	Mickey	0.993813	183	Mon, 19 Sep 2016 22:45:07 (-5.5d)
7	—	A Series Of Unlikely Explanations	0.993721	98	Mon, 19 Sep 2016 23:26:40 (-8.9h)
8	—	idle_speculation	0.993574	6	Mon, 19 Sep 2016 23:46:03 (-6.5d)
9	↑3	Nickel	0.993554	64	Mon, 19 Sep 2016 23:32:22 (-26.7h)
10	↓1	BM (aka BatMan)	0.993544	103	Mon, 19 Sep 2016 23:51:50 (-0.5h)

图 13.16 rank

可以看到前 10 名的预测AUC得分都在 0.99 以上。对于type 7的AUC为 0.9564，还有改进的空间。主要提升的方向如下。

- ❑ 对所有类型的数据进行一轮完整的学习过程。因为type 7数据占总体数据为 (3157)/(904683+490710+429408+207465+157615+4253+3157) = 0.0014367691853286616，即占比才 0.1437%，因此模型的预测性能主要由type1~5决定。
- ❑ 使用多种模型进行训练，最后选择出最优的模型。这里我们仅仅采用GBDT模型，你也可以使用Random Forest模型或者xgboost模型（很多参赛者都使用了该模型）进行训练。
- ❑ 对离散型取值特征group_1和char_10_act进行独热码编码。前面提到，为了降低计算复杂度我们没有对这两列进行独热码编码（此时样本的特征数量在 100 的量级）。为了提高预测预测性能，我们可以对这两列进行编码，此时样本的特征数量将呈爆炸式增长，样本的特征数量都在 10000 的量级。

全书符号

- x：标量。
- $\vec{\boldsymbol{x}}$：列向量。
- $\vec{\boldsymbol{x}}^T$：行向量。
- $\vec{\boldsymbol{x}} = (x^{(1)}, x^{(2)}, \cdots, x^{(n)})^T$：列向量的 n 个分量。
- $\vec{\boldsymbol{x}}_k$：第 k 个样本输入。
- $\vec{\boldsymbol{x}}_k = (x_k^{(1)}, x_k^{(2)}, \cdots, x_k^{(n)})^T$：第 k 个样本输入的 n 个特征。
- $\vec{\boldsymbol{w}}^{<k>}$：$\vec{\boldsymbol{w}}$ 第 k 轮迭代时的值。
- $\boldsymbol{A}^{-1}$：矩阵 $\boldsymbol{A}$ 的逆矩阵。
- $\boldsymbol{I}$：单位矩阵。
- $a\vec{\boldsymbol{x}}$：标量与列向量的乘法。
- $\vec{\boldsymbol{u}} \cdot \vec{\boldsymbol{v}}$：向量的点积。
- $\vec{\boldsymbol{u}}^T\vec{\boldsymbol{v}}$：行向量乘以列向量，结果为向量的点积。
- $\vec{\boldsymbol{u}}\vec{\boldsymbol{v}}^T$：列向量乘以行向量，结果为矩阵。
- $\boldsymbol{A}\vec{\boldsymbol{x}}$：矩阵与列向量的乘法。
- $f(x)$：以标量为自变量的函数。
- $f(\vec{\boldsymbol{x}})$：以向量为自变量的函数。
- $P(\boldsymbol{X} = x)$ 或者 $P(x)$：随机变量 $\boldsymbol{X} = x$ 发生的概率（概率分布函数）。
- $P(\boldsymbol{X} = \vec{\boldsymbol{x}})$ 或者 $P(\vec{\boldsymbol{x}})$：随机变量 $X = \vec{\boldsymbol{x}}$ 发生的概率（概率分布函数）。
- $P(\boldsymbol{X} = x, \boldsymbol{Y} = y)$ 或者 $P(x, y)$：随机变量 $\boldsymbol{X} = x$ 且随机变量 $\boldsymbol{Y} = y$ 发生的概率（联合概率分布函数）。
- $P(\boldsymbol{X} = \vec{\boldsymbol{x}}, \boldsymbol{Y} = \vec{\boldsymbol{y}})$ 或者 $P(\vec{\boldsymbol{x}}, \vec{\boldsymbol{y}})$：随机变量 $X = \vec{\boldsymbol{x}}$ 且随机变量 $Y = \vec{\boldsymbol{y}}$ 发生的概率（联合概率分布函数）。
- $P(\boldsymbol{Y} = y/X = x)$ 或者 $P(y/x)$：已知随机变量 $\boldsymbol{X} = x$ 的条件下，随机变量 $\boldsymbol{Y} = y$ 发生的概率（条件概率分布函数）。
- $P(\boldsymbol{X} = x; \theta)$ 或者 $P(x; \theta)$：随机变量 $\boldsymbol{X} = x$ 发生的概率（概率分布函数）由参数 θ 决定。
- $P(\boldsymbol{Y} = y/X = x; \theta)$ 或者 $P(y/x; \theta)$：已知随机变量 $\boldsymbol{X} = x$ 的条件下，随机变量 $\boldsymbol{Y} = y$ 发生的概率（条件概率分布函数）由参数 θ 决定。